Formalising a Real-Time Coordination Model

A Thesis Submitted to The University of Dublin, Trinity College,
for the degree of Doctor in Philosophy

Colm Bhandal

October 28, 2014

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidate's own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Colm Bhandal

Summary

The subject of this thesis pertains to coordination in systems of entities characterised by mobile, real-time behaviour and unreliable communication over a wireless network. Such systems are now being deployed both in academic trials and in real-world scenarios towards the achievement of various goals e.g. search and rescue, surveillance, object tracking, traffic management. Coordinating the behaviour of entities in these systems is a non-trivial task, due to real-time demands, sensor inaccuracies, wireless coverage unreliability and entity mobility, among other concerns.

A coordination model is a theoretical framework for the development of strategies by which systems of entities can operate safely towards the achievement of their goals. Comhordú is a coordination model that facilitates the definition of scenarios consisting of a number of entities and a safety constraint. Furthermore, it provides a means of deriving behavioural constraints upon entities such that the safety constraint in a scenario is respected while entities pursue their goals. However, there is no formal proof that this is the case. Moreover, the presentation of Comhordú is not formal and is somewhat fragmented, based mostly on a mixture of prose and UML diagrams. Hence there is a need for its formalisation.

There are several benefits to formalising a model. A formal specification avoids the ambiguities often implicit in a natural language description. Also, through the process of formalising a model, often necessary modifications or improvements to the original design become apparent. Furthermore, a formal specification can be subjected to rigorous mathematical analysis, often with the aid of a computer.

This work is the first to address the formalisation of Comhordú. The goal is to produce a formal specification of Comhordú along with a verification of its correctness with respect to maintaining safety. That is, for every scenario that can be developed with Comhordú, are the coordination strategies sufficient to guarantee that safety for that scenario is maintained?

Preceding a formal specification of Comhordú, critical analysis of the original model is provided. This analysis uncovers a number of ambiguities in the original model and argues that certain aspects of the original model would be difficult to formalise. Following this analysis, modifications are made to the original model yielding a revised model that is more amenable than the original to the process of formalisation. These modifications alone are significant contributions of this work. In light of the modifications made, it is necessary to validate this revised model against the original. That is, the revised model is analysed in how closely it matches the original model. In particular, this analysis focuses on elements from the original that are altered or even wholly absent in the revised model.

The revised model is formally specified using a novel formal specification language, developed specifically for the purpose of this work. This language incorporates a general behavioural model for systems of mobile, real-time entities with unreliable wireless communication. Within this language, a protocol is specified; this is based on the idea of behavioural constraints from the original presentation of Comhordú. A proof is provided that this protocol is correct with respect to maintaining safety; roughly speaking the proof says that adherence to the protocol by all entities in a system implies safety in that system. The exact notion of safety is defined later in this thesis.

The aforementioned formal specification and proof are encoded using the proof assistant Coq. The language and protocol description are encoded in Gallina, the specification language of Coq. The proof of correctness is partially translated into Coq. That is, the main theorem pertaining to safety is proved in terms of a number of

lower level results, some of which are assumed without proof in the machine encoding. For those results left unproved in Coq, sketch proofs are provided by hand.

Overall, then, this work achieves the following. The Comhordú model is modified to produce a revised model, which is validated against the original. The revised model is formally specified by means of a novel formal specification language and a term in that language. This formal specification is translated to Gallina, the language of the proof assistant Coq, and the proof of correctness is partially completed using Coq.

Acknowledgements

Firstly I'd like to extend a warm thank you to my supervisor Dr. Arthur Hughes. His positive feedback and encouragement often kept me going when things were really starting to look pear-shaped. I always got a lot out of our meetings, whether if was coffee, obscure mathematical knowledge, or vital feedback. I'm especially grateful for the work he put in towards the end scrutinising the endless walls of text I kept sending him.

To my (unofficial/co) supervisor Prof. Matthew Hennessy, I extend my gratitude for the time he dedicated to meeting me regularly over the past two years. It must have taken a good deal of patience and effort to listen my over-excited ramblings. I could not have finished this without the critical guidance he gave me towards getting my work on track.

I would like to thank Dr. Mélanie Bouroche, the creator of the original Comhordú model, for taking the time to discuss various issues and providing key feedback.

Thanks to Dr. Andrew Butterfield for the domain specific advice. His feedback at important junctures along the way has been essential in steering this work.

I am grateful to the rest of the foundations and methods group here at Trinity College, for listening to my various talks over the years and providing me with feedback and new ideas.

I would like to thank my friends who kindly agreed to do some proof-reading for me: Michael Clear, Paul Laird, Carlo Spaccasassi, Giovanni Bernardi, Mithileash Mohan and Brendan Cody Kenny. Their fresh pairs of eyes spotted things I never would have considered myself.

I'd like to thank Dr. Sarah Loos and her supervisor Prof. André Platzer for their correspondence about the KeyMaera hybrid systems tool. Particularly thanks to Sarah, whose lengthy email conversation partially cured my ignorance of hybrid programs and was a great learning experience for me.

Thanks to Edsko de Vries for the "tcd-phd-thesis.cls" template.

I would like to acknowledge the Irish Research Council as the sponsor of this work.

While this is a formal statement of acknowledgements, it wouldn't feel right if I didn't acknowledge the people who helped me get through this thing on a more personal level. At this point though, there's a slight *cringe*. These things always give the impression of an inflated sense of grandeur on the behalf of the author "Oooooh look at me, I'm so important, all these people exist in the world for the sole purpose of supporting me through whatever mad endeavour I choose to undertake" etc. etc. Hopefully just saying that will neutralise the effect somewhat before I launch into a big list of thank yous that would make any oscar-winning speech look like a one-liner. All joking aside though, I do appreciate every person I'm about to mention here.

To my family, I'd like to extend a huge thanks. Mam and Dad, fair play for putting up with a 26 year old man still living with his parents and eating all their porridge. I appreciate all the subtle and overt support you've given me to complete this thing. Special thanks Dad for the printing and Mam for insisting on washing my bedsheets periodically every 9 days. And the rest of you lot, Liam, Louise, Edel and Rachel. How bloody lucky am I to have you 4 lunatics by *default*? I didn't even have to fill in any forms for you guys. There was no postage-and-packaging fee. You just came free with life and that's pretty cool.

To all my close friends I've met over the years: you guys are so much more than just the "antidote to thesis stress" but you're that too. Cheers guys. Some of those sessions with ye in pubs/houses/apartments towards the end kept my sanity barely hanging on by a thread. And it always made the last-train-home-alone journey that little bit less depressing to see the wacky pictures, videos and comments you posted up on various social media outlets that I'm afraid to mention by name here lest I get sued for copyright infringement.

Thanks to the people in the Lloyd building, just for generally being good company but also probably for specific things that I can't remember now. In particular, to the attendants Chris and Tina thanks for letting me into my office all those times I idiotically locked myself out, and just for all the chats in general. To Shane the hardware guy next door, thanks for fixing my computer that annoyingly decided to start crashing twice daily very close to the thesis deadline. To the various people who have been in and out of room 112 of the Lloyd building over the past five years, I've enjoyed your company and random discussions.

I'd like to acknowledge Trinity college Dublin for the institute of scholarship it provides. Material benefits aside, as a scholar, I've had the chance to meet some wonderful, interesting and eccentric people; usually all three in one. Commons for me has often been an oasis of social energy in an otherwise dull day; isolated in the depths of our singularly focused work, us PhD students probably don't get to interact with people as much as we should.

To the people I met in spring/summer schools, it's been a pleasure to keep the company of the likes of yourselves. The formal methods and theoretical computer science communities are scattered across the globe in little isolated fragments, so it's great when we get together en masse and bring the theory to life. Not to mention all the nerdy jokes that would elude/bore $99.999999\ldots\%$ of the general population.

No doubt I've probably forgotten some person, institute, place, time, animal, theorem or concept that I should have acknowledged. To anyone who really does belong here, but who I've foolishly forgotten to mention, please accept my excuse of "thesis-related stress", and give yourself the pat on the back you know you deserve.

This is a bit unorthodox, but I'd like to acknowledge my self. Yes that's right, if you hadn't inferred my inflated sense of self importance by now here's the hard evidence. To my past self, I forgive you for being such a moron starting into this whole thing without a clue in the world, ignoring valuable advice and just generally mucking about. I also appreciate you eventually copping on, dragging your ass out of bed and just doing the work, day by day. Cheers mate, you got me here so far. To my future self, why are you reading this, was five years of your life at it not enough? Anyway, please don't be too critical of the content.

Finally, I would like to acknowledge my old mug. I had so many cups of tea in that mug. It had the caption *Stress is forbidden* written on it. One day, I left it on top of a soap dispenser and accidentally knocked into it with my elbow. That was the end of my old mug. The irony is, I was probably too stressed to notice it sitting there. The lesson? Heed thy mug. But don't despair! There is a silver lining to this story. I have a new mug with sheep in different moods printed all over it. This new mug has stood by me during the final most hectic months of my PhD and we have become quite close even in this short space of time. Long may I sip tea out of this new mug.

Well, I don't know who is worse. Me for resorting to acknowledging my self and inanimate objects, or you for insisting on reading this far even after the author has clearly shown signs of insanity. In any case, I think it's time to wrap this up. So many thanks again to everyone, and here goes the rest of this thesis.

Contents

List of Figures

List of Tables

As far as the laws of mathematics refer to reality,
they are not certain;
and as far as they are certain,
they do not refer to reality.

ALBERT EINSTEIN[Einstein, 1921]

Chapter 1

Introduction

This thesis presents the formalisation of a coordination model called Comhordú using a process algebra and proof assistant. The original model, initially developed in [Bouroche, 2007], is a framework for designing real-time systems of mobile entities that are in some sense safe. Formalisation of Comhordú incorporates both a precise specification of its components and a verification that the systems it generates are indeed safe. The remainder of this chapter explains the concepts of formalisation and coordination, among other key concepts. It also provides some motivation for the formalisation of Comhordú and summarises the content of the thesis.

This chapter is organised as follows. Section 1.1 motivates and introduces the concept of coordination in real-time systems of entities. Section 1.2 argues in favour of the use of formal methods, mentions some existing formal methods, and provides a simple example of a formal language, along with a specification and some behavioural simulations in that language. Section 1.3 outlines some previously published papers relating to this work. Section 1.4 briefly describes the contributions of this thesis. Section 1.5 outlines the structure of the subsequent chapters.

1.1 Coordination

This work is concerned with a concept called coordination as it applies to a certain class of systems. The systems of interest are those in which physical entities operate in real time, are capable of movement in space and can communicate with each other over a wireless network. These systems span disciplines such as robotics [Dunbabin and Marques, 2012, Mostofi, 2013], autonomous vehicles [Marinescu et al., 2010, Zhu et al., 2009] unmanned aerial vehicles [Peng et al., 2012, Kushleyev et al., 2013, Alvissalim et al., 2012], and any other areas in which autonomous physical agents operate concurrently. The increasing ubiquity of these systems necessitates research into design methods that allow them to operate in a safe and progressive manner. For example, a team of unmanned aerial vehicles might need to survey an area without colliding into each other.

The broad question that comes to mind in this context is, how can systems of entities be *coordinated*? Before this question can be answered, it is necessary to give a rough intuition as to what is meant by *coordination*. Coordination is defined elsewhere as: "the management of interactions both amongst entities, and between entities and their environment, towards the production of a result" [Bouroche, 2007]. Coordination models are theoretical frameworks for modelling distributed systems and developing coordination schemes for these systems. System designers can use these models as a blueprints towards building real systems.

Comhordú is a coordination model developed in [Bouroche, 2007]. It applies to the class of systems previously mentioned i.e. those in which there are physical mobile entities and a wireless network. In such systems, as argued in [Bouroche, 2007] and [Bouroche and Cahill, 2008], coordination becomes non-trivial due to the possibility of network coverage degradation. To address this difficulty, Comhordú introduces a concept called responsibility. This is an alternative to the traditional approach of consensus. With consensus entities await acknowledgements from other entities before committing to a course of action. In contrast, the idea of respon-

sibility is that certain entities in a system are elected "responsible". These entities then independently guarantee coordination within the system, i.e. without necessarily any feedback from neighbours. Responsibility can be transferred to neighbouring entities if a potentially hazardous action is to be taken.

Figure 1.1 depicts consensus among five entities: a sender notifies the other four entities of some action it intends to perform and awaits acknowledgement messages from these entities before proceeding with this action. In the figure, it is assumed that the sender, depicted as a circle at the bottom of the diagram, first sends messages to all of the other four entities, also depicted as circles. These messages are shown as arrows. The replies from other entities, also shown as arrows, are assumed to follow receipt of the initial message. Once the entity receives replies from all entities, it can then safely take action.

Figure 1.2 portrays a transfer of responsibility: the sender warns neighbouring entities of its intention and then these become responsible to adapt accordingly. These transfer messages must be periodically sent as long as an entity remains acting in a hazardous way, because, due to the potential mobility of entities, the set of neighbouring entities is not fixed with respect to an entity.

Coordination in Comhordú is based around the notion of a safety constraint. This is a condition that should never be violated by a system. The coordination schemes proposed in Comhordú are developed towards the aim of maintaining system safety while allowing entities to progress. In Comhordú, a safety constraint language is defined allowing for the specification of scenario-specific safety constraints. These safety constraints are properties based on the state variables of the constituent entities of a scenario. For example, a safety constraint might say that one car should never be travelling towards another car whenever the two are within a certain distance of each other; this would conceivably prevent collisions between the two cars.

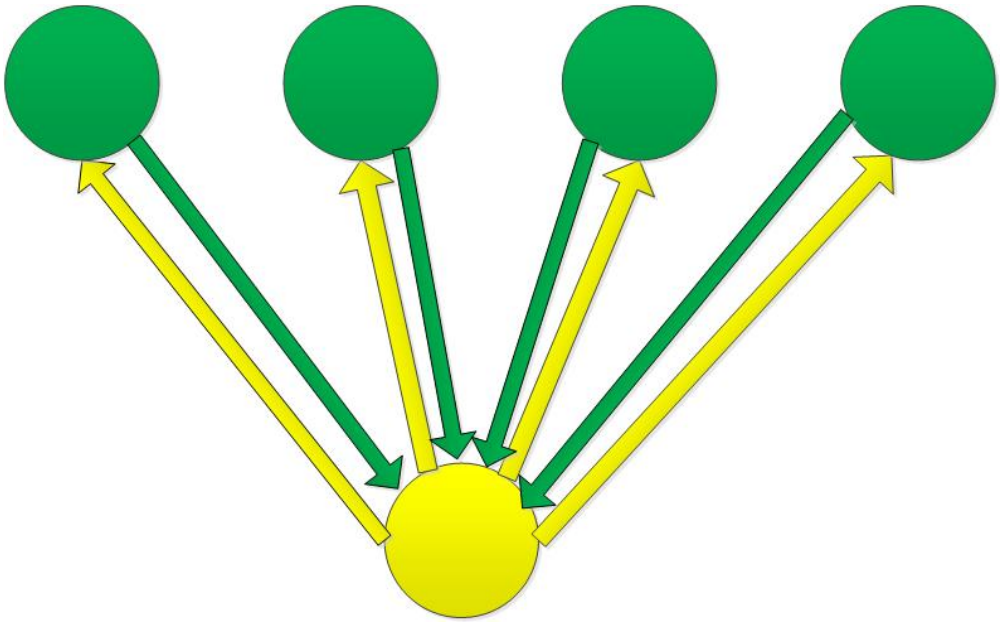


Figure 1.1: The consensus approach. A sender entity (bottom) asks its neighbours for permission to perform a planned action. It then awaits acknowledgement from each of the entities before it proceeds with the action. Entities in the diagram are shown as circles; messages are represented by arrows.

The original presentation of Comhordú does not include a formal basis for its coordination model: "this work however does not present a formal definition of the coordination model, nor a formal proof of its correctness" [Bouroche, 2007]. Rather, what is presented is an informal collection of diagrams and prose, some semi-formal definitions, and informal arguments towards the guarantee of safety provided by the model. This work strives to provide a formalisation of Comhordú. Towards this end, the discipline of formal methods is considered.

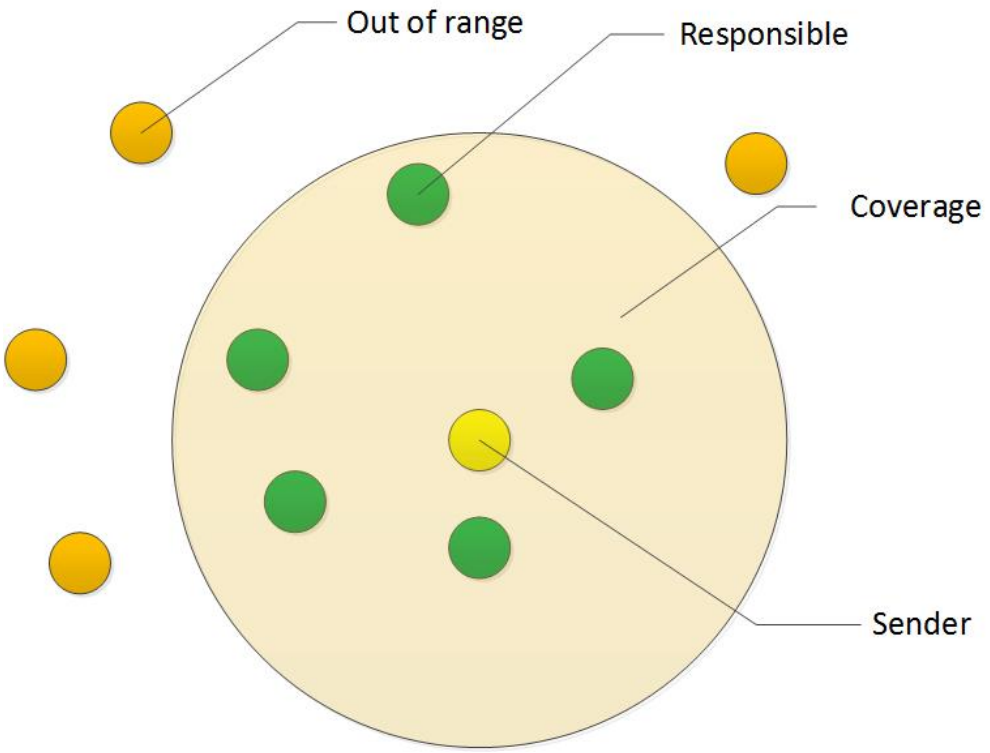


Figure 1.2: The transfer primitive. An entity notifies other entities within a certain region of an action it is performing. It is then the responsibility of the receiving entities to adapt their behaviour in order to maintain safety. Entities in the diagram are shown as circles. The message broadcast is shown as a large translucent circle.

1.2 Formal Methods

Formal methods are "mathematically based languages, techniques, and tools for specifying and verifying ... systems" [Clarke and Wing, 1996]. Some key terminology that pertains to formal methods is now explained; admittedly this is somewhat verbose but is included for sake of the non-formal-methods reader. The process of specifying and verifying a system is referred to as formalisation. The system in question is said to be formalised. A system that is specified using a formal language is said to be formally specified. The result is called a formal specification or just a specification. A formal specification can be augmented by proving properties about it, this is the process of formal verification or just verification. A specification that does not use formal methods is referred to as "informal" or sometimes "semi-formal" if a moderate amount of rigour is employed.

There are a plethora of existing languages for formal specification. Process algebras such as the classic Calculus of Communicating Systems (CCS) [Milner, 1982] are languages for modelling various forms of computation. A "toy" process algebra is given in Example 1.1; following this Example 1.3 illustrates a simple process in that language along with its behaviour. Several recently developed process algebras are geared specifically towards the modelling of broadcast communication [Mezzetti and Sangiorgi, 2006, Godskesen, 2007, Merro et al., 2011, Prasad, 2006, Cerone et al., 2013]. Real-time logics e.g. [Alur and Henzinger, 1994, Chaochen et al., 1991, Cimatti et al., 2009] are languages chiefly intended for the specification of timing properties for systems.

Languages are usually presented by rigorously defining their syntax and then providing a semantics over terms. That is, terms in the language are interpreted as having certain behaviours. The behaviours usually model some form of computation and are often associated with some kind of state-space. That is, a formally specified system tends to be associated with a set of states, possibly infinite, representing all the possible configurations of that system. For example, the state of a computer might be modelled as the values stored at all of its memory locations. The state-space would then be all the possible combinations of values for the memory locations.

There are a number of tools e.g. [Larsen et al., 1997, Platzer and Quesel, 2008, Huet et al., 2007] for machine based specification and (semi-)automatic verification of systems. Systems can be modelled with these tools and automatic algorithms can be run in order to check certain properties or behaviours of a system. The main advantage with such automation is that the processing power of a computer is exploited: computers can search through state spaces that would be intractable to explore by hand. Even still, tools suffer from a phenomenon known as state-space explosion: the state-space tends to grow exponentially relative to the size of the specification.

It is reasonable to ask what is the benefit of going to such lengths to formalise a model? The argument here is that the contributions of such a formalisation are manifold. As stated elsewhere, a motivation for using formal specification is "to add precision, to aid understanding, and to reason about properties of a design" [Woodcock and Davies, 1996]. An imprecise model or design of some system can be difficult to comprehend and may have many interpretations due to inherent ambiguities. This lack of understanding will most likely lead to the design being incorrectly implemented, perhaps even to the point of system failure. Systems whose safety is crucial are in particular need of precise specification: "Many real-time systems are safety-critical, and therefore deserve to be specified with mathematical precision" [Schobbens et al., 2002]. Formalisation also allows for the rigorous assertion of various system properties. Properties can be proved by hand or verified automatically using a software tool.

Example 1.1 (A Simple Timed Process Algebra)**.** A simple process algebra is presented here, to give the non-formal-methods reader an idea of what a process algebra entails. This is a minimal version of timed CCS with some features stripped away. The idea here is to convey the essence of a typical process algebra with real-time behaviour. There is an inductive definition of the language syntax and then a separate definition of the behaviour of the language via what is called a structural operational semantics. The inductive syntactic language definition consists of a number of constructors i.e. functions to build terms of the language. The structural operational semantics defines a relation over the language. It is comprised of rules for deducing when one process can transition to another one.

The syntax of this language consists of four constructors, explained in the following.

- The first constructor is $\mathbf{0}$, which can be thought of as the "base case" for the induction in the definition. It is a term of the language that is supposed to represent a process that does nothing, i.e. a stopped or dead process.

- The next constructor is discrete action prefixing. It takes as arguments a discrete action δ and a process $\mathbf{P}$ and returns a process $\delta.\mathbf{P}$. A discrete action represents something that happens without any time passing. Of course, in reality all actions will take some time to execute but for all intents and purposes some actions can be considered to be instantaneous e.g. the execution of a single command of a program. The meaning of a prefixed process $\delta.\mathbf{P}$ is that it is waiting to do a discrete action δ and then evolve to become the nested process $\mathbf{P}$.

- A special type of prefix called a delay prefix guards a process $\mathbf{P}$ with a time t e.g. in $\varepsilon(t).\mathbf{P}$. The variable t is drawn from some time domain, e.g. the real numbers. The meaning of this process is that no actions of the process $\mathbf{P}$ can be done until the time t has elapsed. The choice constructor is denoted by the plus sign. It takes two processes and returns a process.

- A choice term is capable of "choosing" to perform a discrete action from either of its sub-components and then behaving like that sub-component.

Given below is the inductive language definition. The meta-variables $\mathbf{P}$ and $\mathbf{P}_i$ denote processes. The following definition can be interpreted as giving the rules for building a process $\mathbf{P}$. That is, it formally defines the aforementioned constructors. In the definition, constructors are separated by vertical bars $\mid$. Every term in the language is built by a finite application of these constructors.

$$\mathbf{P} ::= \mathbf{0} \mid \delta.\mathbf{P} \mid \varepsilon(t).\mathbf{P} \mid \mathbf{P}_1 + \mathbf{P}_2$$

Table 1.1 gives a simple semantics corresponding to this language. That is, it furnishes the preceding syntactic terms with concrete behaviours that are rigorously defined via a set of rules. The rules are all of the form $\frac{Premise}{Conclusion}$. This can be read as "Whenever *Premise* is true, *Conclusion* can be derived". Sometimes the premise of a rule is empty, meaning that the conclusion is unconditionally true. The conclusions to these rules are of the form $\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'$, which can be interpreted as meaning that $\mathbf{P}$ transitions to the process $\mathbf{P}'$ and in the meantime performs the action α. Actions α can be either discrete, in which case they are denoted using the meta-variable δ, or they can be timed, in which case they are denoted with a meta-variable d, for delay. It is assumed delays are strictly positive values in the time domain, in comparison to time values t prefixing terms which may be 0.

The rule DO says that a process $\delta.\mathbf{P}$ prefixed by a discrete action δ is capable of doing that action and transitioning to the nested process $\mathbf{P}$. The next two rules, CHOICE-L and CHOICE-R, characterise behaviours for choice terms. If two processes $\mathbf{P}_1$ and $\mathbf{P}_2$ are combined together via the choice construct, and either can perform a discrete transition, then the entire process can perform that transition with the same action and derivative process. Note that in the course of taking a discrete transition either the left or right sub-component of a choice construct is "chosen", so to speak. That is, the process behaves as if the other sub-component didn't exist at all. There is no discrete transition possible for the process $\mathbf{0}$, characterising the fact that this process is incapable of performing any discrete action. The rule TIMEOUT says that if a process is guarded by a time of 0 then its actions are free to be performed. Note the use of the meta-variable α in this rule, denoting either a discrete or a timed action.

The remaining rules pertain to delay actions. DEL-NIL and DEL-PFX allow the nil and action prefixed processes to delay by an arbitrary amount and remain unchanged after the delay. This can be thought of as the processes waiting. The rule DEL-SUB allows a delay-prefixed process to delay by a certain amount d and this amount to be subtracted from the delay guard in the resulting process. Notice in this rule the condition $d \leq t$ which restricts the delays from occurring if they exceed the value of the guard. The rule DEL-CHOICE says that if two processes can delay by the same amount, then the sum of these processes can also delay by that amount. Notice that this does not force a choice to be made between the pair. That is, only discrete actions cause the choice construct to "collapse", so to speak.

Table 1.1: Simple discrete semantics of a toy language.

DO	$\dfrac{}{\delta.\mathbf{P} \xrightarrow{\delta} \mathbf{P}}$
CHOICE-L	$\dfrac{\mathbf{P}_1 \xrightarrow{\delta} \mathbf{P}'}{\mathbf{P}_1 + \mathbf{P}_2 \xrightarrow{\delta} \mathbf{P}'}$
CHOICE-R	$\dfrac{\mathbf{P}_2 \xrightarrow{\delta} \mathbf{P}'}{\mathbf{P}_1 + \mathbf{P}_2 \xrightarrow{\delta} \mathbf{P}'}$
TIMEOUT	$\dfrac{\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'}{\varepsilon(0).\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'}$
DEL-NIL	$\dfrac{}{\mathbf{0} \xrightarrow{d} \mathbf{0}}$

Continued on Next Page...

Table 1.1 – Continued

DEL-PFX	$\dfrac{}{\delta.\mathbf{P} \xrightarrow{d} \delta.\mathbf{P}}$
DEL-SUB	$\dfrac{d \leq t}{\varepsilon(t).\mathbf{P} \xrightarrow{d} \varepsilon(t-d).\mathbf{P}}$
DEL-CHOICE	$\dfrac{\mathbf{P}_1 \xrightarrow{d} \mathbf{P}'_1 \qquad \mathbf{P}_2 \xrightarrow{d} \mathbf{P}'_2}{\mathbf{P}_1 + \mathbf{P}_2 \xrightarrow{d} \mathbf{P}'_1 + \mathbf{P}'_2}$

That concludes this short example. In Example 1.3 the behaviour of a concrete term in this language is explored. For a more thorough introduction to process algebras, the reader is referred to the textbook on reactive systems [Aceto, 2007]. ▲

Remark 1.2 (Environment Delimiters)**.** For the remainder of this thesis the following delimiters are used to mark the end of an environment. Remarks such as this one and examples such as Example 1.1 are ended with a black triangle ▲. Definitions are ended with a black square ■. The standard Q.E.D. symbol □ is used to mark the end of a result such as a theorem or lemma. ▲

Example 1.3 (A Simple Process)**.** Recall the simple process algebra from Example 1.1. Below is an example of a simple process in that language. The purpose of this process is to repetitively perform the action *out* three times. The actions are separated by a period of 1 time unit. That is, before each execution of the action *out*, at least 1 unit of time must pass. Also, an action *stop* can happen before any of the *out* actions occur. This might model, say, an external event causing the process to abort its execution prematurely. The process is modelled in a somewhat repetitive manner using an interleaving of delay prefixes and action prefixes.

$$\varepsilon(1).out.\varepsilon(1).out.\varepsilon(1).out.\mathbf{0} + stop.\mathbf{0}$$

The process operates as follows. Initially, the only possible actions are the passage of time and the *stop* action. If the *stop* action is ever performed, then the process evolves to $\mathbf{0}$ i.e. it stops. Otherwise, up to 1 unit of time can pass; this may occur as a single delay action or as a sequence of delays. At any point between consecutive delays, the stop action may happen in which case the process transitions to $\mathbf{0}$. Once 1 unit of time passes, the left hand term has timed out, and the *out* action is ready to be performed. From here, either more time can pass, a *stop* action can occur, or the *out* action can be taken. In the case of the *out* action being taken, the left hand term is "chosen" and the stop action is no longer available. From then on, only delays and more *out* actions are possible, until the process $\mathbf{0}$ is finally reached.

Sequences of transitions can be used to demonstrate the behaviour of this process. Each individual transition can be deduced using only the rules of the semantics given in Table 1.1. A transition always contains a source process, an action and a derivative process. Sequencing transitions, such that the derivative process of one transition is used as the source process of the subsequent transition, yields what is called a trace, and can be represented by a sequence of processes interleaved with the corresponding actions. Abstractly, a trace consists of processes $\mathbf{P}_i$ and actions α_i and takes on the following form.

$$\mathbf{P}_1 \xrightarrow{\alpha_1} \mathbf{P}_2 \xrightarrow{\alpha_2} \mathbf{P}_3 \ldots \xrightarrow{\alpha_n} \mathbf{P}_{n+1}$$

Let us now explore some traces demonstrating the behaviour of the above process. Each transition in these traces can be deduced from the rules of Table 1.1. However, for the sake of brevity, an explicit deduction for each transition is not given. Rather it is left to the curious reader to infer where each transition came from according to the language rules. It is believed that for the traces given here this should be a straightforward enough task.

To begin with, let us imagine that the stop action never occurs and the left hand side process executes all of its *out* actions as soon as they are ready to be performed. In the following trace, a delay of 1 time unit occurs, followed by an execution of the *out* action. This happens again. The last *out* action is then preceded by two delays adding up to 1 time unit, and the final term is $\mathbf{0}$.

$$\begin{aligned}
\varepsilon(1).out.\varepsilon(1).out.\varepsilon(1).out.\mathbf{0} + stop.\mathbf{0} &\xrightarrow{1} \\
\varepsilon(0).out.\varepsilon(1).out.\varepsilon(1).out.\mathbf{0} + stop.\mathbf{0} &\xrightarrow{out} \\
\varepsilon(1).out.\varepsilon(1).out.\mathbf{0} &\xrightarrow{1} \\
\varepsilon(0).out.\varepsilon(1).out.\mathbf{0} &\xrightarrow{out} \\
\varepsilon(1).out.\mathbf{0} &\xrightarrow{0.6} \\
\varepsilon(0.4).out.\mathbf{0} &\xrightarrow{0.4} \\
\varepsilon(0).out.\mathbf{0} &\xrightarrow{out} \mathbf{0}
\end{aligned}$$

The availability of a discrete action does not imply that it must be performed immediately. This is facilitated by the rule DEL-PFX, which allows a prefixed process to delay indefinitely. The following trace illustrates this point. First a delay of 1 time unit happens, rendering the left hand process capable of performing the *out* action. However, it is supposed that another delay of 11 time units occurs, showing that actions do not need to happen as soon as they are ready. From here, it is possible that any number of further delays occur. In this short example though, let us say that the *stop* action occurs next. This might indicate that some sort of an error has precluded the execution of the left hand side process.

$$\begin{aligned}
\varepsilon(1).out.\varepsilon(1).out.\varepsilon(1).out.\mathbf{0} + stop.\mathbf{0} &\xrightarrow{1} \\
\varepsilon(0).out.\varepsilon(1).out.\varepsilon(1).out.\mathbf{0} + stop.\mathbf{0} &\xrightarrow{11} \\
\varepsilon(0).out.\varepsilon(1).out.\varepsilon(1).out.\mathbf{0} + stop.\mathbf{0} &\xrightarrow{stop} \mathbf{0}
\end{aligned}$$

Intuitively, this process specification seems somewhat wasteful. Specifically, there is much repetition in the sequence of delays and *out* actions. As a human programmer, it is natural to see such a specification and ask if there is some form of abbreviation possible to alleviate any of this repetition. For example, a C++ programming background would suggest the use of some variant of a "for loop" to achieve the repetition. Furthermore it is impossible in this toy language to specify a process that carries on repeating its actions indefinitely. This is because all process are in some sense finite i.e. they all eventually terminate in their behaviour.

Another potential point of contention arising in relation to this process is the fact that the *stop* action, representing some sort of program error, can only happen at the start of the process. That is, once the left hand term begins executing, the *stop* action is no longer available. It would make more sense to have the *stop* action available at all points in the execution of the program. However, to model such behaviour in this language requires a verbose specification, involving a series of nested *stop* actions for every *out* action. Later in this thesis, a more expressive language is presented which facilitates cleaner expression of such behaviours.

The reason behind the choice of this particular example is that it relates somewhat to the previously mentioned notion of responsibility; specifically it models periodic broadcast in quite an abstract way. The message being broadcast is abstracted with the action *out* and the period of the broadcast is 1 time unit. Also, while the broadcast is periodic, only a finite number of broadcasts can be modelled in this language; in this case 3 broadcasts are modelled. Later in this thesis, a process with indefinite broadcasting capability is specified using a language of greater complexity to this one.

▲

1.3 Publications Related to this Thesis

At the time of writing, there are two peer-reviewed publications in relation to this work. These are detailed below.

- **An Abstract Model of a Coordination Protocol using the UPPAAL Model checker.** In this paper [Bhandal et al., 2011a] the UPPAAL modelling tool is used to generate an abstract model of Comhordú and some properties about the model are verified.
- **A Process Algebraic Description of a Temporal Wireless Network Protocol.** This paper [Bhandal et al., 2011b] presents a description of a protocol using a process algebra. The protocol characterises the concept of responsibility that is intrinsic to Comhordú.

1.4 Contributions

Recall that the goal of this research is to formalise the Comhordú coordination model and verify that it is correct. Initially it would seem then that a research question corresponding to this goal would be: "is Comhordú correct?" However, attempting to answer this question reveals that it is not as simple as it appears. This is because the original presentation of Comhordú is not formally defined. Rather, it consists mostly of a fragmented collection of English descriptions and UML diagrams. To answer this question definitively, then, it is necessary to first provide a rigorous formal definition of Comhordú and of what is meant by "correct".

However, this matter is complicated by the intractability of formally modelling certain aspects from the original model. The infeasibility of the inclusion of these aspects leads to the development of a number of modifications to the original model in order to render a revised model that is more amenable than the original to formalisation. These modifications raise the issue of validation: how well does the revised model adhere to the original?

Taking into account the above points, the research questions addressed by this thesis are as follows.

- What revisions need to be made to Comhordú prior to its formalisation?
- How does the revised model relate to the original formulation of Comhordú?
- Is the revised model correct, i.e. does the safety condition hold in the revised model?

To answer these questions, this work provides the following contributions.

- Several modifications are made to the original Comhordú model. The modified model is a contribution in itself. Overall, it consists of less components than the original model but it is more precise and so better adapted for formalisation.
- A formal modelling language is developed, embodying the behaviour of real-time systems of mobile entities. It consists of a number of sub-languages for modelling different aspects of this behaviour. The languages assume a process-algebraic style i.e. terms are defined using an inductively constructed syntax and behaviour is defined via an operational semantics.
- A protocol is devised and specified in a software fragment of the aforementioned language. This protocol is a novel feature of the formal model, though it is based on the notion of responsibility from the original Comhordú model.
- A proof demonstrating the correctness of this protocol is provided. That is, the protocol is shown to guarantee safety, in a certain sense discussed later.
- The aforementioned language, protocol and proof are encoded into the proof assistant Coq.

It is stressed here that while this work is based on the original Comhordú, the development of a revised model, modelling language, protocol and proof are all novel to this work.

1.5 Summary of Content

The remainder of this thesis is broken down as follows:

- Chapter 2 is a literature review of the areas of distributed systems and formal methods.
- Chapter 3 investigates the feasibility of formally modelling Comhordú. In the process of this investigation, revisions are made to the original model in order to render it more amenable to formalisation.
- Chapter 4 introduces a formal specification language for real-time systems of mobile entities.
- Chapter 5 defines a protocol in the aforementioned language and explores its behaviour.
- Chapter 6 gives an overview of the main results comprising the safety proof.
- Chapter 7 describes the main features of the Coq development of this model, and outlines some pending work to be done in Coq. At the time of writing, the Coq model of Comhordú can be found online at: https://www.scss.tcd.ie/ bhandalc/CoqDoc/toc.html.
- Chapter 8 reviews the work, provides some concluding remarks and discusses future developments for Comhordú.

Chapter 2

Literature Review

The goal of this thesis is to address the problem of formalising a coordination protocol called Comhordú. The research area pertaining to this goal is cross-disciplinary: Comhordú belongs to the distributed systems domain whereas formal methods constitute a field in their own right. Both areas need to be studied for different reasons. On the one hand, distributed systems provides a context and motivation for Comhordú. On the other hand, formal methods allows for the exploration of solutions to the problem of formalising Comhordú. It is also important to study the intersection of both fields i.e. the application of formal methods to distributed systems. The fields of distributed systems and formal methods are covered here. Related work in the application of formal methods to distributed systems is left for Section 8.4.

Figure 2.1 roughly depicts the research area in question. Arguably there are other disciplines involved and alternative ways of categorising the various sub-disciplines. For example, hybrid systems research can be considered a field in its own right, with formal methods forming a sub-discipline. However, from the perspective of this work, the diagram shown seems to capture best the relevant areas of research. The two broad fields are distributed systems and formal methods. Within each of these are shown sub-themes e.g. robotics in the distributed systems domain. The intersection of the two disciplines represents the field of applied formal methods to distributed systems. The formalisation of Comhordú, which forms the subject of this thesis, can be said to lie in this intersection. Nonetheless, this thesis is approached primarily from a formal methods perspective, building upon existing work in the distributed systems domain. Accordingly then the remainder of this chapter focuses in some detail on formal methods whereas distributed systems is given more of an overview.

This chapter is broken into sections. At the end of each section is a discussion exploring the relevance of the literature presented in that section with respect to the current thesis. The first two sections give background to the Comhordú model, while the final three contain formal methods theory that applies to the problem of formalising and verifying Comhordú. Section 2.1 explores some research in distributed systems, providing a context and motivation for the Comhordú model and protocol. Following this is a chapter solely dedicated to future cities and vehicular computing research, which is arguably a sub-discipline of distributed systems but warrants discussion in its own right because it is more directly related to Comhordú. Following this in Section 2.3 is a presentation of formal methods theory, with a particular focus on process algebras for the specification of systems. Real-time formalisms, which seem more specifically related to this work, are given separate attention in Section 2.4. Finally, there is a discussion of formal methods tools in Section 2.5.

2.1 Distributed Systems

Broadly speaking, the Comhordú model applies to what may be referred to as distributed systems. This is quite a general term, spanning a number of areas. This section gives a brief overview of distributed systems research related to Comhordú. Subjects covered here include wireless sensor networks (WSN), transport, unmanned

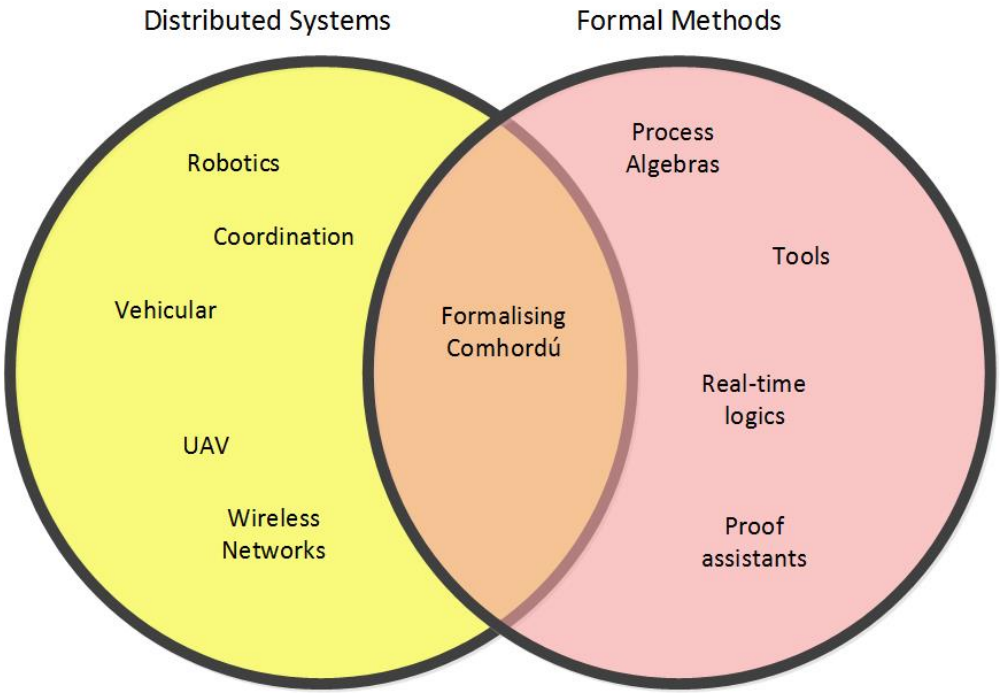


Figure 2.1: Rough depiction of the research area. The subject of this thesis is formalising Comhordú, which lies in the intersection between formal methods and distributed systems.

aerial vehicles (UAV), coordination, robotics, and multi-agent systems. Since there is an overlap between these subjects in the literature, they are not all considered individually in the discussion to follow. This area gives only a brief glimpse of distributed systems literature, as the primary focus of this work is formal methods. For a more comprehensive background of the work inspiring Comhordú, the reader is referred to [Bouroche, 2007].

Unmanned aerial vehicles (UAV) constitute an emerging discipline in distributed systems. Much of the research in this field focuses on a specialised scenario in which the vehicles operate, usually towards the completion of some task or the testing of some algorithm. In most cases, the focus is on the cooperation of multiple UAVs constituting what is known as a swarm. The need for coordination between multiple UAVs is expressed in [How et al., 2004]: "Future UAV teams will have to autonomously demonstrate cooperative behaviors in dynamic and uncertain environments". In [Vries and Subbarao, 2011] a steering technique for swarms of UAVs is proposed and tested using a simulator, with an emphasis on obstacle avoidance. A search and rescue team of UAVs and unmanned ground vehicles (UGV) is given in [Weaver et al., 2013]. In [Wei et al., 2013], a UAV team is proposed for the purpose of "enemy territory reconnaissance, remote area surveillance and hazardous environment monitoring." A similar application of UAV teams is given in [Alvissalim et al., 2012], in which UAVs swarm and self-deploy to restore a telecommunication link that has gone down due to some disaster. Some work is focused more on the dynamics of the vehicles themselves, such as reducing the size of the quadrotors to improve agility [Kushleyev et al., 2013] or using air currents to boost altitude and extend flight duration [Cobano et al., 2013].

Robotics is another discipline that belongs here. Much like UAVs, robotics systems usually involve coordination between the individual entities therein. Often to achieve this coordination a wireless sensor network is employed such as in [Shue and Conrad, 2013]. In [Yasuda et al., 2012], a new method is proposed in which robots adapt to potential danger in their environment and coordinate accordingly. Sensor fusion is presented in [Wagatsuma et al., 2011] as a method for robots to coordinate their behaviour; robots share data collected from cameras to build a view of the environment. Three separate schemes for coordination in mobile robot systems are given in [Jiang et al., 2011]. The schemes are evaluated in terms of their performance using a simulator called "network simulator 2". Coordination in a surveillance application is investigated in [Trigui et al., 2012]. Here three approaches to coordination are proposed: centralised, distributed and market-based. The robots in

question are given the goal of target tracking and capturing.

Cyber-physical systems CPS, discussed in [Rajkumar, 2012], bridge the gap between computing and physical machines. In this paper, massive advancements in human society are predicted due to the emergence of this area. Of interest here are the safety and timeliness requirements that the paper predicts will be essential to such systems: "Since a CPS interacts with the physical world, it must operate dependably, safely, securely, efficiently, and in real time".

2.1.1 Distributed Systems: Discussion

The purpose of the preceding exposition of distributed systems research is to provide a context and motivation for the Comhordú protocol. The context is characterised by a number of sub-fields the focus of which is the development of coordination methods in various different types of systems. The motivation for Comhordú in this setting is the need for a general protocol for coordination that is robust with respect to fluctuations in network coverage. However, the overall goal of this thesis is not to motivate Comhordú itself but rather argue towards a need for its formalisation and then provide such a formalisation. Thus this section is minimal, providing just enough detail for a reader grasp the basics of the background behind Comhordú. For a more thorough argument motivating Comhordú itself, as opposed to its formalisation, the reader is referred to [Bouroche, 2007].

2.2 Future Cities and Vehicular Computing (FCVC)

Future Cities and Vehicular Computing (FCVC) is concerned with the application of computer science to vehicle navigation and traffic management towards the ultimate goal of improving transportation infrastructure, particularly in cities. While this discipline may be considered to fall under the banner of distributed systems, it is considered here in its own right because the research herein is directly related to Comhordú. The following list discusses research in this domain.

Comhordú and SEM Of primary relevance here is Comhordú, the coordination model and protocol at the heart of this thesis. The model is presented in [Bouroche, 2007] as part of a PhD thesis. An abridged version can be found in [Bouroche and Cahill, 2008]. The basics of Comhordú are as follows. A system of mobile "entities" is assumed: these could be cars, robots, unmanned aerial vehicles, more or less anything covered in 2.1. These entities are capable of communicating over a wireless network. It is assumed that the entities want to achieve some goals and in doing so must behave in potentially unsafe modes of operation relative to each other. In this context, the goal is coordination: to allow the entities to achieve their goals while maintaining overall system safety. The challenges of this goal are outlined in [Senart et al., 2006]. The Comhordú protocol addresses these challenges in a novel way compared to pre-existing methods of coordination, which are primarily consensus-based, using a notion called responsibility. The claim made in [Bouroche, 2007] is that if all entities operate according to Comhordú, then the system is guaranteed to be safe. However, this claim is not formally verified, nor can it be, unless the entire Comhordú model is formalised. Hence, the work in [Bouroche, 2007] leaves open the need for formalisation, which is addressed in this thesis. For more detail on the Comhordú protocol, see Section 5.2.

Underlying Comhordú is an assumed lower-level model called the Space-Elastic Model (SEM), which is the subject of a PhD thesis [Hughes, 2006]. In this model, network coverage is defined as the area about an entity to which messages can be delivered within some time bound. Since wireless communication is in general unreliable, this spatial coverage fluctuates with time, hence the term "space-elastic". What is guaranteed by SEM is that whenever a message is sent by an entity at a certain time, then within a bounded time from the moment of delivery of that message, a notification will be given to the sender informing it of the coverage to which its message was sent. The efficacy of SEM is explored using a

test scenario involving a pedestrian crossing in [Bouroche et al., 2006]. The details of this model are summarised in Section 3.1.

Intelligent Traffic Much of the work in the FCVC group is based around improving traffic in some way, referred to here as "intelligent traffic" research. When compared to Comhordú, traffic is more restrictive and so in a certain sense easier to abstract. For example, cars can be assumed to stay in lanes and so position can be abstracted by the lane a car is in and a single number representing where that car is on the lane. The opportunity for abstraction afforded by the restrictions of traffic, e.g. lane adherence, facilitates more concise formal models, but their applicability under these abstractions becomes limited to a particular scenario.

In [Asplund et al., 2012], a similar protocol to Comhordú called CwoRIS is verified using the Z3 theorem prover. The protocol allows cars to enter a road intersection without collisions i.e. it ensures mutual exclusion of cars from the intersection. Again in a similar vein to Comhordú, responsibility is employed to ensure safe behaviour: "every entity is responsible for making sure that it does not enter an unsafe state with respect to any other entity". The work is domain-specific to the case of the intersection, unlike Comhordú, which is intended for wider application.

In [O'Hara et al., 2012], a simulation platform called MDDVsim is presented. It combines microscopic details, e.g. the speeds and positions of the individual vehicles, with macroscopic behaviour e.g. traffic flow density. This platform is tested on a scenario with two cars side by side on a two lane highway. It is argued here that simulation can save time and money by allowing certain vehicular behaviours to be tested without actually constructing them. This resonates somewhat with the advantages of using formal methods, although the latter is generally more time consuming and more rigorous.

A body of work addresses the problem of improving traffic flow on motorways. Some challenges associated with this problem are mentioned in [Cahill et al., 2008]: "real-time hierarchical scheduling, local real-time coordination and real-time inter-vehicle communication." Also in that work, the benefits of "vehicle scheduling" are given as the reduction of "air pollution, traffic congestion, stress and accidents". The slot based approach [Marinescu et al., 2010] is one step in this direction, with the aim of optimising traffic with intelligent vehicles. The scheme proposed is that vehicles drive in virtual slots that move along the motorway with time thus reducing traffic congestion and giving drivers better time bounds on their journeys. The paper includes two algorithms: one for slot acquisition and another for lane merging. The core idea is that because vehicles drive (semi)-autonomously in slots that have a predetermined behaviour (speed profile, trajectory), smaller gaps can be left between vehicles and a form of cooperation can be achieved, compared with the competitive form of driving seen on the roads these days. Extending this work is [Marinescu et al., 2012], in which the on-ramp traffic merging is added to augment the previous merging algorithm. Another method for traffic congestion reduction appears in [Arnaout and Bowling, 2011]. The method is tested using a microscopic agent-based simulation upon which quantitative statistical analysis is performed e.g. flow rate and average speed are measured.

A location tracking system for a city, called HERO is proposed in [Zhu et al., 2009]. The system consists of a number of nodes that track the locations of vehicles as they move around the city. An overlay network matching the road infrastructure is augmented with a top-level hierarchical network to allow vehicle position information to be propagated across space. Such a network can facilitate applications such as traffic diversion. It is also conceivable that safety protocols similar to CwoRIS [Asplund et al., 2012] could be achieved using this network.

In [Lattner et al., 2005], vehicles are given a "knowledge base" and "common sense" for making high level "intelligent" decisions. The vehicles construct a qualitative representation of their surroundings and use this to make decisions about their behaviour. One limitation of this work is scalability in the number of objects in the vehicles' environment.

Overall, intelligent traffic research usually focuses on a specific scenario e.g. lane merging or junction scheduling, allowing various abstractions to be employed. Related work in this domain appears in Section 8.4, with the added theme of formal verification.

Wireless Networks The communication medium used by Comhordú systems is a wireless network. While the Comhordú model includes the network as a rather abstract component allowing broadcast messages to be sent, it is worth exploring at least some literature on networks, particularly in relation to FCVC research. In [Asplund and Nadjm-Tehrani, 2012] a new scheme is proposed for analysing the "routing latency" of messages in the network. Similarly in [Asplund, Mikael Nadjm-Tehrani, 2012] a number of theoretical results are given in relation to the worst-case latency of broadcast in a network. This is closely related to the notion of message latency in Comhordú, as can be seen from the introduction: "an important aspect [of time critical applications] is to be able to guarantee message delivery within a certain amount of time". A medium access control (MAC) protocol for "reliable and timely broadcast", called the Real-time Reservation Protocol (RRP) is presented in [Zhang et al., 2013]. Here "periodic beaconing" i.e. the repetitive sending of broadcast messages is employed to achieve reliable broadcast. A notion called staleness is introduced as part of the protocol, which measures the time since receipt of the last beacon for a particular entity. All of this work on wireless networks is quite closely related to SEM, i.e. the communication protocol underlying Comhordú. However, no further exposition of network-related research is needed here since it is beyond the scope of this work.

2.2.1 FCVC: Discussion

In one sense, Comhordú "belongs" in the FCVC group. Its original motivation was to achieve coordination in systems of entities such as traffic on roads. However it is argued here that for a number of reasons Comhordú is both more general and less applicable to traffic-based scenarios than the related work already discussed. This is not to say that Comhordú is *unsuitable* in such applications, rather that the more tailored approaches seem better suited.

In support of this point, notice that Comhordú is not restricted to any spatial paths. That is, while it is reasonable to assume that traffic always remains in a single lane on a road, Comhordú allows for a most flexible general movement of its entities. This vastly increases the applicability of Comhordú to domains outside of FCVC relative to any protocols that take single lane movement as a given. However it becomes questionable whether Comhordú would be useful at all in a scenario where single lane movement is a reasonable assumption to make. That is, there seems to be no need to use Comhordú when a more tailored approach can probably deliver better results.

There are also other application-specific abstractions other than single-lane movement exploited by the above work that cannot be factored into Comhordú because of its generality. For example in [Asplund et al., 2012], a junction is represented as an abstract atomic component and safety can be reduced to mutual exclusion of entities from this shared resource. Such abstractions cannot be made in Comhordú, which physically models only space itself and the constituent entities as points in that space.

Also much of the work found in this FCVC field is more low-level than Comhordú in that network specifics are taken into account such as message loss and MAC layer protocols. While these considerations are useful for implementations of the underlying SEM model they are beneath the scope of the Comhordú model itself.

At this point it may be asked, for what kind of systems is Comhordú a good modelling choice? It seems that for any systems with restricted movements, such as traffic or railway systems, the power of generality offered by Comhordú is wasted; it seems in these cases, tailored approaches exploiting the constraints on movement are more suitable. This immediately suggests that Comhordú is ideally a model for systems with little or no restrictions on spatial movements. Examples of such systems are underwater guided vehicles, unmanned aerial vehicles, or indeed ground vehicles without any road infrastructure. These applications are further explored in Section 8.5.2.

2.3 Formal Specification Languages

This section is chiefly devoted to the consideration of formal languages for the purpose of specifying the Comhordú model. Formal languages are well-defined mathematical entities that facilitate the exact specification of systems. Usually, a language of this kind has a formal syntax defined using mathematical notation and a formal semantics interpreted over a mathematical structure e.g. a graph. They can be viewed as a subset of the more general field of formal methods, which are defined as "mathematically based languages, techniques, and tools for specifying and verifying . . . systems" [Clarke and Wing, 1996]. The application of formal methods usually involves using some language to specify a system so that it can be rigorously verified as "correct" in some sense. The verification may be supported by a tool but most of the discussion on these is left to Section 2.5.

The merits of formalisation are manifold, and are discussed extensively in the literature. Here a cross-section of these pro-formal ideas taken from papers in a number of different fields are combined to give a thorough argument in favour of formalisation. We begin with a point made in the artificial intelligence literature outlining what is arguably the most immediate benefit of formally modelling a system: "[Formal specification] allows a precise and non-ambiguous description of the architecture" [Hilaire et al., 2010]. A similar point is made in relation to formal methods in robotics: "Formal specification necessitates to state requirements precisely. A beneficial side effect is that it focuses discussions and manifests design decisions" [Walter et al., 2010]. Yet again almost the same point is made in the hybrid systems literature: "Formal approaches may help with a deep analysis that takes care of the precise semantics of the requirements" [Cimatti et al., 2012]. In other words, simply the act of formalising a system alone is already advantageous in that it makes the specification of the system precise, and "forces our hand" in making design decisions that might otherwise be ignored.

Such ignorance at an early design stage can have negative consequences later on. As argued in [Florian et al., 2012]: "The correctness of such [real-time] systems is of significant importance because malfunctions can induce notable economic costs, and they can even cause the loss of human life." The argument involving safety is also stressed in [Schobbens et al., 2002]: "Many real-time systems are safety-critical, and therefore deserve to be specified with mathematical precision." Formal methods is also hailed in the field of hybrid systems: "rigorous reliability analysis requires formal modelling" [Henzinger, 2000]. Furthermore, "[f]ormal verification becomes more and more important as computerized control systems in safety-critical systems grow significantly in complexity" [Platzer and Quesel, 2008]. Put another way, many systems, particularly those with real-time involved, are safety critical i.e. the consequences of them malfunctioning are catastrophic, potentially even fatal. Thus, there is a strict need for these type of systems to operate correctly, and the rigour offered by formal methods can help achieve guaranteeing this correct operation, which is difficult to achieve otherwise given the ever increasing complexity of such systems.

Formal modelling usually entails some sort of verification, though as discussed already the specification of a system formally is contribution enough as it is. One of the advantages of formally verifying some property is rigour that is otherwise unachievable: "It is very easy to overlook some detail when carrying out by hand even an elementary derivation. Informal arguments for equivalence of processes are error prone, since exhaustive case analyses on possible transitions are elusive" [Honsell et al., 2001]. Not only are the state spaces too vast for human exploration, there are also some subtleties that tend to escape our intuition: "For a complicated language, it soon goes beyond human capabilities to keep track of the consequences of design-decisions in the semantics and one can often overlook possible counter-intuitive phenomena introduced there" [Mousavi and Reniers, 2006].

Substantiating these claims is that fact that "[d]uring the last ten years, formal verification of digital systems has evolved from an academic subject to an approach accepted by the industry" [Fränzle and Herde, 2006]. That is, formal methods is not just something that looks good on paper, it has been tried and tested in many real-world applications. However, the level to which industry has adopted formal methods is questionable: "its take up in industry has been meager" [Hunt, 2012]. This suggests the existence of drawbacks to formal methods that cannot be ignored.

To start, there is the issue of validation: does the formal model indeed capture the system it intends to represent? It seems as though the answer to this question is never a resounding "yes" i.e. "absolute validation is Utopian" [Ciaffaglione and Honsell, 2003]. The reason for this is usually the loss of detail that cannot be avoided through the process of abstraction. For example, "[n]on-realistic vehicle dynamics, simplified communication models and idealistic localisation can all detract from the credibility of evaluations carried out" [O'Hara et al., 2012]. In general, it is hard to know how to abstract "correctly", and indeed a "possible difficulty is in justifying the abstractions made during the modelling process" [Webster et al., 2011]. In fact, [Bertolino et al., 2012] claims that "the job of validating and verifying [systems is] extremely difficult".

Another "important obstacle is generating faithful models from system description formalisms, in particular for mixed software/hardware systems" [Calude, 2009]. In other words, specifications should not deviate too much from the real-world behaviour of the system they intend to model. Still, "[v]ery often, theory makes assumptions oversimplifying reality" [Calude, 2009]. This is not just a formal methods issue, but applies to any theory whatsoever that hoes to model the real world.

The remainder of this section reviews a number of different formalisms in terms of their applicability in formalising and verifying the Comhordú model and protocol. These formalisms span a wide range of themes including two-tiered or multi-tier languages, languages for networks, hybrid systems and process algebra, among others. These themes are not mutually exclusive, and so the following organisation is not to be taken strictly. What can be said is that the majority of the formalisms considered are geared towards the modelling of networks. A large amount of these network-oriented formalisms, particularly the more recent ones, adopt what can be called a "two-tiered" approach, in that they separate the modelling of "nodes" and "processes", in some sense representing the physical entities and the software components in the network respectively.

In the following, first process algebras in general are discussed in Section 2.3.1: these are formal languages that have an algebraic flavour and form the basis of most of the work appearing in this section. After this, the two-tiered languages already mentioned are presented in Section 2.3.2. Then Section 2.3.3 covers other network-based languages. Following this hybrid systems are discussed in Section 2.3.4; these are systems that combine continuous behaviour with discrete behaviour.

2.3.1 Process Algebra

Process algebras are mathematical constructs usually geared towards the modelling of software or hardware. They can be thought of as abstract or simplified programming languages, with certain implementation details omitted such as platform-specific commands or specialised libraries. The advantage of abstracting in this way is that the languages are simple enough to be subjected to analysis and verification techniques. For example, properties of programs written in such languages can be specified and proved. Many of the formalisms examined in the following, particularly in Section 2.3.2 and Section 2.3.3 can be classified as process algebras. This section covers the "classical" process algebras, laying the groundwork for the more modern ones which are presented next.

A good introductory text for process algebras is [Aceto, 2007]. Introduced in this book is one of the oldest and best-known process algebras called the Calculus of Communicating Systems (CCS). Also included is the timed version of CCS called TCCS, a logic called Hennessy-Milner logic for specifying properties of processes and another formalism called Timed Automata (TA). This text is highly recommended for anyone interested in familiarising themselves with process algebra.

Milner's Calculus of Communicating Systems (CCS) [Milner, 1982] is one of oldest and most inspirational process calculi. Many existing languages are strongly influenced by CCS. The language of CCS is a simple, inductively built syntax which includes action prefixing to model sequential execution and parallel composition to model concurrency. The communication in CCS is point to point and synchronous. Processes communicate with each other by performing complementary actions, which may be thought of as input and output. In classical CCS, the distinction between input and output is arbitrary. When CCS is augmented to value-passing CCS as in [Bergstra et al., 2001a], the output action is more obviously distinguished from the input by the fact

that it is the output process that determines the value sent, while the input process simply listens for this value, accepts it, and binds it to a variable that may occur in the process body.

Timed CCS [Yi, 1991] extends the original syntax with one feature, the delay prefix, and adds delay transitions to the semantics. The delay prefix essentially models a process which has "gone to sleep", to borrow a term from multi-threading in an imperative software programming. The prefixed process can do nothing but delay until the time in the delay prefix has elapsed. The delay transition relation, which is added in the timed semantics, models the passage of time in the behaviour of processes.

Another classical process calculus is Tony Hoare's Communicating Sequential Processes (CSP). A good introduction to this language can be found in [Davies and Schneider, 1995]. Also found in this paper are extensions of this language to include time. The main deterrent to using CSP or one of its variants to model Comhordú is the way communication is handled. The problem is that "CSP is based on a multiway [event] synchronisation mechanism, but it does not differentiate between input and output" [Ene and Muntean, 2001]. Now, at first sight event synchronisations seem promising for modelling broadcast, in that they allow multiparty participation in an event. For example, let us say one process wishes to send a message and all others wish to receive it. It might be possible to model both sending and receiving as participation in an event that symbolises the transmission of the message. However, this idea runs into trouble because of conflicts. For example, consider two processes trying to send the same message. If the transmission of this message is modelled as an event, and sending/receiving this message are both modelled as participation in this event, then two senders can send the same message at the same time. This violates our intuitive notion of broadcast, i.e. that of a single sender and multiple receivers.

The process algebras discussed up to now are "classical" languages, but they do not seem suitable for a natural description of Comhordú. In particular, the features of broadcast and mobility are missing, and some "hack" would be needed in order for these to be included, which would no doubt complicate the model somewhat. The question then is, does there exist a process algebra suitable for modelling Comhordú? Or indeed, can this be achieved by some other formalisms other than process algebra? To answer this a number of more modern process algebras are first considered in Section 2.3.2 and Section 2.3.3, then other formalisms are explored in the subsequent sections.

2.3.2 Two-tiered Network-based Languages

A plethora of process calculi have recently emerged to deal with the modelling of wireless networks. Many of these employ what is commonly referred to as a "two-tiered" approach, where the nodes of a network and the software components running within those nodes are given separate treatment. That is, the languages are split into two separate "sub-languages", one for nodes and communications and the other for software and internal computations. In a certain sense, the nodes can be seen as the hardware components of a network and the processes running within them the software components. The advantage is a separation of concerns, making systems not only easier to understand but also more amenable to proofs: "a traditional process calculus must intermix the computation of neighborhood information with the protocol's control behaviour. This tends to render such models unnatural and unnecessarily complex" [Singh et al., 2010].

To begin with, let us consider the Calculus of Wireless Systems (CWS) [Mezzetti and Sangiorgi, 2006]. This is a language for modelling systems at the data link layer- the lowest communication layer. Systems are modelled on two levels:

1. There is a fixed network of nodes i.e. nodes cannot be created or destroyed dynamically. Each node has a position in space and a fixed broadcasting radius. Position and broadcasting radius do not change over time for a given node. The nodes represent the physical transmitter/receiver apparatuses that handle data transfer.

2. Within each node runs a local sequential process that performs internal computations. This process can communicate with its environment only indirectly through its host node.

A separate semantics is given for processes and networks of nodes. The process semantics is simple and similar to CCS. The main focus of the network semantics is on modelling message collisions. Although useful for modelling at the datalink layer, this calculus seems too detailed for use on the Comhordú model, which abstracts from collisions by relying on the underlying Space-Elastic Model (SEM). Perhaps this underlying model could be formalised using CWS or its timed variant TCWS, but that is beyond the scope of this work. Furthermore, though it is suggested in the paper as future work, dynamic position and broadcasting radius updates are not featured in CWS. These would be essential for a Comhordú model, which relies on the notions of adaptive coverage and spatial mobility of agents. A final drawback of this language with respect to the modelling of Comhordú is its lack of time, but there is now a timed version of the language, TCWS, which is discussed next.

The Timed Calculus of Wireless Systems, TCWS [Merro et al., 2011], is strongly based on CWS [Mezzetti and Sangiorgi, 2006]. This calculus introduces a notion of time. Broadcasts are modelled by begin and end actions and take a certain amount of time based on the value being broadcast. Time is modelled discretely by the concurrent execution of the σ action by all nodes in a network (and all processes within each node). Network topology is given by neighbour sets: Each node is linked to all nodes in its neighbour set. Only linked nodes will hear each other i.e. broadcasts will only reach nodes that are linked to the sender.

For similar reasons to those given for CWS, TCWS is unsuitable for modelling the Comhordú protocol. The addition of time in this version of the language is primarily an aid to model message collision in even more detail than before. Also, time is modelled discretely: it seems that it would be difficult to encode the real-time properties of Comhordú using only discrete time. As previously argued, a message collision model is too detailed for this work since collisions are not included in the Comhordú model.

In [Godskesen, 2007] the Calculus for Mobile Ad Hoc Networks (CMAN) is proposed. Like all languages in this section, it is two-tiered. A logical location is given to nodes in the network and the connectivity of the network is modelled via a neighbour set. A Timed Calculus for Mobile ad hoc Networks (TCMN) is given in [Wang and Lu, 2012]. An interesting feature of this language is that locations are modelled abstractly in the sense that all that is assumed is a distance function between any two locations. This leaves the dimensionality of the space unconstrained. Node movement is modelled by migration, a single step event by which a node changes its location. Arguably this is an unrealistic abstraction by which nodes essentially have infinite speed i.e. no time passes during a switch in location. Perhaps the reason for this is that time is discrete, and so can't be arbitrarily divided based on the size of a jump in location. While this seems like the most suitable language of those considered for the purpose of modelling Comhordú, it is still somewhat lacking. The model of time is not continuous and the communications model is too low-level.

In [Prasad, 2006], a preliminary calculus for Mobile Broadcasting Systems (MBS) is presented. The novel idea of this calculus is that there are processes and there are rooms. Much like the nodes of CWS [Mezzetti and Sangiorgi, 2006], rooms comprise the top level system, while processes execute within rooms. Unlike CWS however, processes are given the freedom to walk between rooms i.e. they are granted mobility. The room is analogous to the broadcasting radius of CWS or the neighbour set of TCWS [Merro et al., 2011] in that speech by a processes is heard by all processes in the same room as that process, and only by those processes.

Initially, it appears as though the mobility feature of MBS would be sufficient to encode agent mobility in the Comhordú model. Each room could model a cell in space- perhaps a unit square, and processes moving through space would be modelled by processes walking among rooms. Of course, space in this case would be discretised and intra-cell mobility would be unobservable. Herein lies the problem. The obvious solution which is to make the discrete space sufficiently fine, i.e. to make the cells sufficiently small, would not work because with a reduction in cell size comes a forced reduction in agent broadcasting range. Furthermore, coverage range would be fixed to whatever room an agent occupies, whereas in the Comhordú model coverage should vary.

Also, in a wireless network such as that featured in the Comhordú model, coverage is not necessarily transitive, as it is in the room model. With rooms, disjoint sets of agents are all connected to each other, which leads to a transitivity property: If agent A is connected to agent B and B to C, then A is connected to C. This is obvious since agents are connected iff they share residence in a room. Such a transitivity property is

characteristic of an Ethernet with rooms as machines on the Ethernet. However, the property certainly does not hold for a wireless ad-hoc network. Such networks operate on the basis of possibly indirect communication i.e. the sending of a message through multiple parties from sender to receiver.

The remaining calculi are all similar to those already mentioned, with minor variations. One of these is the ω calculus [Singh et al., 2010] in which network topology is modelled with a graph structure. For the Calculus of Mobile Ad Hoc Networks (CMN) [Merro, 2009], broadcast range is modelled as a radius, locations are modelled in space and mobility is encoded as a discrete jump in space. The calculus CBS# [Nanz and Hankin, 2006], which is a derivative of the Calculus of Broadcasting Systems (CBS), is similar to the others but is geared more specifically towards security analysis, which is beyond the scope of this work. In [Cerone et al., 2013] a language called the Calculus of Collision-prone Communicating Processes (CCCP) is given. This language is chiefly concerned with the modelling of collisions and is similar to TCWS. In [Delzanno et al., 2010] networks are represented as tuples of nodes and connectivity is represented as a graph. This idea of a network as a tuple is adopted in the network language that is used to model Comhordú, presented in Section 4.4.

While these calculi all provide more tailored features towards the modelling of networks than the "classical" process calculi that appear in Section 2.3.1, they are all still lacking somewhat as candidates for modelling Comhordú. Yet more languages are considered in Section 2.3.3.

2.3.3 Other Network-based Languages

In this section a number of other (non-two-tiered) languages are examined. These languages are more dated than those previously mentioned, which seems to show a recent trend towards the two-tiered approach. The novelty of these languages relative to their predecessors is their support of broadcast communication as a primitive. Older languages lack this feature of broadcast: "higher level communication schemes, like broadcast or multicast are encountered in many applications and programming models, but they remain nevertheless poorly represented in the algebraic theory of distributed systems" [Ene and Muntean, 2001]. Furthermore: "it appears difficult to encode broadcast in calculi based on point-to-point communications" [Ene and Muntean, 2001]. That is, broadcast languages seem to fulfill a genuine modelling need that cannot be achieved using other languages.

One of the earliest broadcasting languages is the Calculus for Broadcasting Systems (CBS) [Prasad, 1995]. It is reasonably similar to CCS except with broadcast as the communication primitive rather than handshake (point to point) communication. A key distinguishing feature of broadcast from handshake communication is that broadcast is non-blocking i.e. a sender can transmit a message regardless of who is listening. This is in contrast to handshake communication, whereby a sender and receiver must both be ready simultaneously before a communication can take place. This calculus is further developed and studied in [Hennessy and Rathke, 1995]. A key addition made to the language there is the introduction of what are called "discard" actions, by which a listener may ignore a message rather than input it. This facilitates the possibility that a message may not have reached every entity in the system.

The Timed Calculus of Broadcasting Systems (TCBS) [Prasad, 1996] builds on CBS by introducing time. It has the following properties:

- There are no discard actions. Discard actions are desirable in that they distinguish between a process which ignores a spoken value and one which consumes it.
- Choice is only allowed between one input prefixed process and one output prefixed process. This is a restrictive feature in terms of the processes that can be modelled in the language.
- Delay guards only guard output actions- this is to keep the model input enabled. It is also to induce the emergent properties of persistence and readiness. In a modified version of the semantics [Hennessy and Rathke, 1995] however, a law exists allowing delay prefixed components to discard values rather than input them, and the process P in $\varepsilon(d).P$ is completely guarded by the delay prefix for a strictly positive

d. By guardedness what is meant here is that actions of the nested process are unavailable to the guarded process.

2.3.4 Hybrid Systems

Hybrid systems are those that combine the discrete behaviour of software components with the continuous behaviour of "real-world" objects. The study of hybrid systems, particularly their formalisation, is a rather recent pursuit. According to [Calude, 2009] it is still the case that "[w]e badly need holistic rigorous design approaches taking into account the interaction of mixed software/hardware systems with their physical environment". A similar argument is given in [Cimatti et al., 2012]: "Existing languages are too limited to represent requirements of safety-critical applications, because they do not capture both the continuous evolution of physical entities over time and the discrete changes of actions". This seems like an overstatement, since there are hybrid modelling languages in existence. However it does shed light on a genuine immaturity in the field of formalisation of hybrid systems in comparison with other fields of formal methods. Much of the research on hybrid systems is devoted to tackling the problem of exploring the massive state spaces induced by the inclusion of continuous dynamics.

A key paper in this area is [Henzinger, 2000], which is concerned with classifying automata based on their receptiveness to various flavours of algorithmic analysis. A language called CHARON is introduced in [Alur et al., 2000] allowing for modular, reusable, hierarchical specifications of hybrid systems. In [Alur et al., 1994], some algorithmic techniques from Timed Automata literature are modified slightly and used for the purpose of verifying linear hybrid automata. These are a sub-class of hybrid automata which are restricted in that their continuous behaviour is in some sense linear. In [Cimatti et al., 2012] Satisfiability Modulo Theories (SMT) techniques are used in order to validate systems in terms of their requirements.

While the theory of hybrid systems is certainly elegant, the languages therein generally lack features supporting the modelling of wireless networks, such as those found in Section 2.3.2 and Section 2.3.3. Also, the focus of hybrid systems research seems to be on small "toy" examples that are amenable to automatic algorithmic analysis e.g. model checking. Due to the exponential increase in the state-space of these systems relative to the systems size, known as the state explosion problem, it is debatable whether these examples are scalable to real-world applications. This will be further investigated in Section 2.5 which covers tools for the modelling and analysis of systems, including hybrid systems.

2.3.5 Formal Specification Languages: Discussion

Let us now synthesise the formal theory presented thus far into a discussion relating it to the goal of formalising and verifying the Comhordú model. In short, the languages investigated are all somewhat lacking in this regard. Still, with some slight modifications, much of the theory visited can be applied to the task of formally modelling Comhordú. The following outlines the shortcomings of existing languages and concludes by mentioning some features of the languages that can be salvaged for the purposes of this work.

To begin with, it is postulated elsewhere that "the adopted formal language must be as close as possible to the natural language used to write the requirements" [Cimatti et al., 2012]. Bearing this in mind, none of the available languages appear to meet the steep demands of expressiveness imposed by Comhordú. To elaborate on this point, a number of key features of Comhordú are outlined in the following along with an examination of how well these features are supported in the languages reviewed.

- Dense time: it is desirable for a language to include a model of time with arbitrarily fine intervals. An example of this would be the rational numbers: given any rational number r there is always another rational number r' within any bound e of r. Another example of a dense time domain is the real numbers. A major setback is that most languages do not incorporate a dense model of time. Often time, if it is included at all in a language, is modelled sparsely e.g. in the language TCWS. A sparse model of time, in

contrast to a dense model, enforces a minimum bound on the size of time intervals that can be expressed i.e. there is an atomic unit of time.

- Location: it should be possible for a language to describe physical entities in space. To the contrary, many languages are primarily intended for the specification of software systems. This is particularly true of earlier languages. While most recent network languages presented in Section 2.3.2 include some notion of location, sometimes this is a logical rather than a physical location e.g. CMAN or the ω calculus. Since physical locations are an inextricable element of the Comhordú model, it is not enough to simply model logical locations.

- Mobility: any language chosen to model Comhordú should have a capacity for modelling the movement of entities over time. Of the few languages that incorporate a notion of mobility, it is generally too abstract for Comhordú e.g. the rooms model of CBS allows for jumps between discrete locations which cannot faithfully capture movement in space. The calculus CMAN allows mobility, but it is unconstrained i.e. an entity can jump from any point in space to another point without any time passing. This precludes reasoning based on upper bounds on speed that is essential to Comhordú.

- Broadcast: one-to-many communication between entities should feature in any language intended for modelling Comhordú. This precludes the use of many of the older languages e.g. CCS. Still, most of the network-based languages presented in Section 2.3.2 and Section 2.3.3 support broadcast communication as a primitive.

- Space-Elastic Model (SEM): no existing languages include SEM directly in their semantics. Still, it is necessary to integrate this essential element into any specification of Comhordú. Thus with any existing language, in order to model Comhordú, a model of SEM would also need to be specified. Initially this does not seem problematic, but on closer inspection the behaviour of SEM appears difficult to encode using only standard language primitives. In particular, it is not obvious how coverage notifications would be modelled.

Of course, the direct exclusion of some or all of these features from any particular language does not immediately warrant its abandonment. It is conceivable that a feature might be implementable within the existing features of a language even though it is not included as a primitive. That is, there might be a "hack" to encode such a feature. However, for the features outlined above this option does not seem viable without an overly complicated specification. For example, SEM could be modelled by dynamically spawning "notification" processes every time a message is sent, with these processes sending a message back to the sender after a set amount of time. However, this significantly complicates analysis of the system since the number of these "spawned" processes varies over time.

Another reason for departure from these languages for modelling Comhordú is that they tend to be too detailed for this purpose. For example, hybrid systems are geared more towards the modelling of specific system dynamics e.g. a bouncing ball or a steam boiler. On the contrary, the exact dynamics of entities in Comhordú is intentionally left unspecified in order to render the model as general as possible. Another example of superfluous detail in languages is the inclusion of low-level details such as message collisions e.g. [Cerone et al., 2013]. Such details are of no use for the purpose of modelling Comhordú, which is a relatively high-level model.

A point that may be made about the papers reviewed under the banner of "formal specification languages" is that many are quite theoretical in their nature. A typical paper presenting a process calculus will allocate some space for the presentation of a language syntax and semantics before devoting most of its content to the exploration of mathematical properties of the language. Often as part of these explorations, equivalence relations are defined over the language and various theorems are proved about them. These developments are interesting in their own right from a theoretical perspective. They are also useful in developing confidence in the "sanity" of the language. However, for the purposes of this work it is more useful to consider language applications rather than these theoretical properties. Hence this section focuses only the basic language features.

While none of the languages considered here have been adopted for the task of formalising Comhordú, many ideas found in their theory have been salvaged in the formalism that is used. This is a multi-tiered language and is given in Chapter 4. It combines many features of CCS with TCBS and also draws upon the idea of a two-tiered language that pervades much of the recent literature on network-based languages.

2.4 Real-Time Formalisms

Real-time is a commonly occurring theme in the formal methods literature. A justification for the development of formal methods is that "[m]any real-time systems are safety-critical, and therefore deserve to be specified with mathematical precision" [Schobbens et al., 2002]. There are many formalisms in existence that are specifically geared towards the modelling of these systems and their properties. While the more general field of hybrid systems is better suited to Comhordú, it is still quite immature, whereas there is a well-established base of literature pertaining to real-time formalisms. Still, "Correct and efficient implementation of general real-time applications remains by far an open problem" [Tesnim Abdellatif, 2010].

There are a number of papers surveying this area. A key survey discussing temporal and modal logic is given in [Emerson, 1990]. These logics are languages for the expression of properties that can be interpreted over structures e.g. traces. Linear temporal logic (LTL) is interpreted over structures in which time progresses deterministically, i.e. in a straight line, hence the term "linear". In this logic it is possible to express properties such as "For all subsequent states in the trace, event A will never happen" or "It is possible for event A to happen at some future stage". Computation tree logic (CTL) is slightly more complex and is interpreted over a more general tree structure i.e. one in which there can be many futures of a given state. To accommodate the increased complexity of the data structures involved, i.e. branching as opposed to linear structures, the logic includes path quantifiers. These allow for statements such as "There exists a path upon which all states obey some property A". The notion of time in these logics is abstract i.e. all that matters is event ordering rather than duration.

More expressive logics allowing for the statement of real-time properties are surveyed in [Alur and Henzinger, 1992]. This paper makes the distinction between the data structure over which the logic is interpreted, called the model, and the logic itself. Models can be further classified according to a number of criteria e.g. whether time is discrete or continuous or whether co-occurring actions can be interleaved without changing the overall behaviour of the system. Real-time logics over such models then extend or supplement the standard temporal logics of [Emerson, 1990] to include real-time constraints. This can be achieved either by inclusion of an explicit global clock, by bounded temporal operators i.e. those with upper and lower time bounds added, or by a feature known as freeze quantification in which a variable is added to a formula quantifier and its value is bound or frozen within the scope of the quantification to denote the time from which the formula is meant to hold.

Timed process algebras differ from timed logics in that they are geared more towards the specification of a system rather than its properties, though the line between the two is not clear-cut. In [Nicollin and Sifakis, 1992] a number of timed process algebras are reviewed. One question that is asked here is what properties are important for a timed language to exhibit? A selection of commonly occurring properties that are often deemed desirable are:

- Time determinism: the passage of time is deterministic. That is, if a system evolves by allowing a certain amount of time to pass, then there is only one possible successor state. In contrast, with a non-deterministic a.k.a. branching time model a system can in general evolve by the same time delay to a variety of successor states.

- Time additivity: This could also be referred to as transitivity. It says that if a system can evolve from state A to state B by some time t_1 and again evolve from state B to state C by some time t_2, then it is possible for the system to evolve in one single delay of length $t_1 + t_2$ from state A to state C. While this

property seems "obvious" of any real system, it is necessary to show that this is indeed the case for any formalism hoping to model such a system.

- Freedom from deadlock: This says that a system will never be "stuck" i.e. unable to evolve from any state. A related property that is harder to define is called livelock, which is when a system is always capable of evolution but can get stuck within a closed pattern of behaviour.

- Maximal progress: Whenever a system is capable of performing a discrete action, then time cannot pass. In other words, all discrete actions are performed before time is allowed to pass. This means that as soon as an action becomes available, it will be executed, which is usually a desirable property to have in a language. The exact meaning of maximal progress depends on the choice of discrete action over which it is defined. E.g. internal computations may be chosen as the actions in question whereas input/output "half" actions may be excluded from the definition. This makes sense in that processes waiting to do a half action generally need to synchronise with another concurrent process with the complementary action, and so should allow time to pass as they wait.

- Persistence: A persistent language is one in which the passage of time does not affect the ability to perform an action. For example, if persistence holds for a language, then a process that is willing to input a value now will be able to input that value after delaying by any amount of time.

- Finite variability: this is the notion that there is an upper bound on the amount of events that can occur within any duration. A related notion is non-zenoness, which is discussed in Section 4.1.8.

In [Ostroff, 1999], a real-time system is described as a parallel composition of two distinct parts: the plant and the control. The plant consists of environmental factors while the control is some piece of software to ensure correct operation of the system. The dual language approach is discussed here, and also in [Furia et al., 2010]. This is the notion that a specification of a system is written in a different language to its implementation. With this approach, formal proofs can be given to show that implementations respect specifications. The alternative is a "homogeneous correctness proof", where both the specification and the implementation are written in the same language and are shown to be behaviourally equivalent with the right hiding operators applied to the implementation.

Recent reviews of this area are [Wang et al., 2004] and [Furia et al., 2010]. The former gives a thorough survey of the area including models of system behaviour, specification languages, tools and comparisons of state-space reduction techniques used for verification algorithms. The latter, being the more recent, also reviews some hybrid systems research and the techniques used therein. It is split into two parts. The first part lays the groundwork by introducing general notions like modelling language and various ways of classifying different languages. The second part actually steps through the various different languages that exist in the literature.

The remainder of this section discusses a small sample of the real-time formalisms that are covered in the aforementioned survey papers, namely those that are in some way relevant to Comhordú. First some temporal logics are examined in Section 2.4.1. Following this are languages for real-time specifications in Section 2.4.2. Some miscellaneous work in the field is reviewed in Section 2.4.3.

2.4.1 Temporal Logic

The logics presented in the aforementioned surveys, namely CTL and LTL, may be referred to as temporal logics. Another logic called the Temporal Logic of Actions (TLA) [Lamport, 1994] treats programs abstractly as state transformers i.e. mappings from an input state to an output state. However, while there is some notion of time involved in these logics it is quite abstract in that only the ordering of events is taken into account. That is the actual durations of times involved are not expressible.

In the early to mid 90s, a number of logics emerged for expressing real-time properties. One of the first of these was the calculus of durations [Chaochen et al., 1991], a real-time logic whose terms are interpreted over intervals. In this logic, the integral operator "counts" how much time is spent in some state during an

interval. The logic is demonstrated through an example of a gas pump. Another real-time logic is Timed Propositional Temporal Logic (TPTL) [Alur and Henzinger, 1994]. The novel feature of this is the use of a freeze quantification operator, which allows a variable to record a certain time instant so that it can be used as a reference point in formulae. The argument for this approach is that it gives all the "intended" formulae and is elegant, whereas previous approaches allow for nonsensical formulae.

A number of axiomatisations are given for temporal logics. That is, systems of axioms are used to completely encode the logics. An example is [Schobbens et al., 2002], in which axiomatisations are given for two logics: event clock logic and metric temporal logic. A restricted version of LTL without any past tense operators is encoded into the Coq proof assistant in [Coupet-Grimal, 2003]. Formulae are defined over coinductive objects called streams, which are essentially infinite lists. In [Pnueli and Kesten, 2002], a deductive proof system is given for the logic CTL*, which is a variant of Computation Tree Logic (CTL). The argument is that proofs give a better insight into why a property is true vs. automated techniques such as model checking. In [Henzinger et al., 1991], two proof systems are given for two different styles of real-time requirements specification: explicit clock approaches and bounded operator approaches. The former involves introducing variables as is the case in freeze quantification. The latter involves adding upper and lower bounds to temporal operators to allow for specification of properties such as "Property A will hold within x units of time."

A recent logic called HRELTL has emerged for the validation of hybrid systems with respect to their requirements [Cimatti et al., 2009]. The focus here is on the "requirements engineering" process, that is the process of turning natural language specifications formal ones.

An example of a simple temporal logic is introduced in Example 2.1.

Example 2.1 (A Toy Demonstration of Temporal Logic)**.** In this example let us envisage a simple temporal logic defined over sequences of natural numbers. The syntax of the logic consists of the inductively defined set of formulae given below. A temporal logic formula is denoted by F. This can be either P, which we assume here to be a predicate over natural numbers, or $F_1 \upsilon F_2$, which can be read "F_1 until F_2".

$$F ::= P \mid F_1 \upsilon F_2$$

The logic is interpreted over sequences. The formula P is true of a sequence exactly when the predicate P holds for the first value of that sequence; this is the standard interpretation for most temporal logics in the literature. The meaning of "until" is that the first formula F_1 holds for some possibly infinite prefix of the sequence and as soon as it does not hold, F_2 holds. A stronger version of "until" asserts the existence of a state satisfying F_2 but here we allow an infinite sequence of F_1 states also.

Let's say there is some predicate over natural numbers Q_2, such that only the number 2 satisfies this predicate. Then the following sequence satisfies this predicate interpreted as a temporal logic formula in this toy logic, because the first state of the sequence is 2.

$$2, 4, 8, 6, 10, 16, 22, 13 \ldots$$

To make things a bit more interesting, imagine a predicate E, which is true for all even number and another predicate Q_{13} which is true for the number 13 only. Then the sequence given above satisfies the composite temporal logic formula $E \upsilon Q_{13}$. That is, the sequence is even until the number 13 is reached. With this definition of until, allowing for infinite sequences of the first property, the below sequence consisting of all even numbers would also satisfy the formula $E \upsilon Q_{13}$.

$$2, 4, 6, 8, 10 \ldots 2k \ldots$$

This is the case even though the number 13 is never reached. For a stricter version of until asserting the existence of the second property, the formula would not be satisfied by this sequence.

This logic is a minimal example intended just to give a flavour of the structure of temporal logics. It is here

given neither formal semantics nor a complete definition involving more operators for greater expressiveness. Also, there is no notion of real-time given here. For these features, the reader is referred to one of the surveys such as [Emerson, 1990] for temporal logic or [Alur and Henzinger, 1992] for real-time logic. ▲

2.4.2 Real-time Languages

A number of real-time languages other than the logics appearing thus far exist. A landmark paper [Alur and Dill, 1994] in formal real-time literature introduces the theory of Timed Automata. Timed Automata are structures for the description of real-time systems. A timed automaton consists of a set of locations, transitions, invariants and clocks. Clocks record the passage of time, but can be reset individually so they may at any given time record different values. The transitions themselves contain a source location, a target location, a guard, an action and a set of clocks to be reset. A transition can only be taken if the guard is true. If a transition is taken, the clocks mentioned in the transition are reset and the automaton is deemed to perform the action of the transition.

Other real-time formalisms take the form of process algebras. For example there is the Algebra of Timed Processes (ATP) [Nicollin and Sifakis, 1994]. An interesting feature of this language is that maximal progress is an emergent property of the semantics. In [Hennessy and Regan, 1995], a process algebra is presented which also has maximal progress. In this language time is treated rather abstractly by augmenting the discrete semantics with a single-tick action, rather than allowing delays to occur over some arbitrary time domain as is done in other approaches from this period. However, with this abstraction, timing properties are quite restrictive. Timed CSP [Davies and Schneider, 1995] is also a timed process algebra. There is no distinction made in CSP between input and output actions.

2.4.3 Other Real-time Literature

The remainder of the real-time formal methods theory reviewed here does not fit into any of the above categories. In [Ostroff, 1999], compositional design methods are given for real-time systems geared towards the tackling of the state-space explosion problem. The advantage of compositionality is that components are simpler to verify individually. The overall system is then proved correct by construction. The proposed method is demonstrated with an example of a reactor-shutdown system using a toolset called StateTime which has a theorem-prover/model checker back end. In [Kesten et al., 2000] Clocked Transitions Systems are introduced, extending Timed Transition Systems and building upon the work of Timed Automata. An interesting contribution of this work is a verification method for checking if a system is non-zeno i.e. if a system is free from infinite behaviour in bounded time. In [Tesnim Abdellatif, 2010] an abstract model is given for software in which timing properties are omitted. On top of this a physical model is built, adding delays to the actions of the abstract model. This allows the separation of concerns between modelling the software and the platform upon which it is running, allowing platform independent properties to be proved. An extension of the SPIN model checker to accommodate real-time periodic software systems appears in [Florian et al., 2012]. It claims to be the first tool of its kind to provide direct support for these type of systems.

2.4.4 Real Time: Discussion

Since time is an integral part of Comhordú it makes sense to study real-time formalisms in the interest of formalising and verifying Comhordú. However, for similar reasons to those given in Section 2.3.5 these languages are deemed unsuitable for modelling Comhordú. To avoid repetition these reasons are not reiterated here. Just to give an example though, most of these languages do not incorporate any notion of broadcast nor do they include any notion of position or mobility.

Moreover, many of the formalisms discussed in this section are primarily geared towards property specification rather than the modelling of systems. While this seems to preclude the adoption of such languages for modelling Comhordú itself, it is conceivable that a real-time logic might be of use in the proof of safety for

Comhordú. This would constitute a dual-language approach, the notion of which is introduced in Section 2.4. Indeed, the original intention was to use some of the existing theory on real-time logic to build the proof. However, this approach proved impractical and real-time logic was abandoned in favour the current approach, which is expounded in Chapter 6.

Despite this, there are elements of real-time systems that have inspired some of the work done as part of this thesis. Most notable of these are certain language properties such as persistence, time determinacy, maximal progress etc. These are visited in Section 4.1 in relation to the software fragment of the multi-tiered language that is used to model Comhordú. Furthermore, Timed Automata notation inspires the illustrations of the protocol appearing in Chapter 5.

2.5 Tools

Arguably the most promising aspect of formal methods is the ability to subject formal specifications to analysis by machines. A number of tools have been developed to this end. Usually, the tools automatically search the state-space of a system to check that some property holds or to generate a counter-example if it doesn't. The underlying algorithms and formal specification language that a tool uses are typically studied in the literature in their own right. Tools usually have a front-end for user-friendly specification of systems; this may be graphical or language-based. Many tools, particularly real-time or hybrid tools, deal with infinite-state systems. For these, there is a need to convert the infinite state exploration problem into some finite symbolic method. Most of the papers on tools focus on the various merits of the underlying theory e.g. language expressiveness, complexity of the search algorithms etc.[1]

A key approach underlying many of the tools in this area is model checking. The historical context within which model checking emerged is portrayed in [Clarke, 2008]. The technique emerged to address the problem of concurrent program verification. Model checking means checking whether some structure is a model for a formula. That is, does the formula hold for the structure? The structures in question are usually the state-spaces of various systems. The massive ongoing challenge faced by the model checking community is devising means to address what is known as state-space explosion. That is the state-space of a system tends to grow exponentially relative to the size of the specification of the system, and so algorithms must do better than simply exhaustively check all states for various properties. Still, even with state-space reduction techniques, model checkers are limited in the size of the systems they can handle, which is their biggest disadvantage.

Another approach involving the use of machines to check systems is automated theorem proving. A survey giving the technical details of this can be found in [Plaisted, 2014]. Theorem provers, rather than check all states of a system, will try to reduce a certain statement to known truths a.k.a axioms via accepted rules of inference. A theorem prover will often use a heuristic to guide it in trying a number of different deduction/reduction steps until it has completed the proof. The resulting proof is often called a proof tree, whose leaves are known results and whose root is the theorem in question, with rules of inference linking children to parents in the tree. It is in fact more correct to call the proof a directed graph rather than a tree because a "child" can in fact have multiple "parents". Fully automated theorem proving runs into a similar problem to model checking: the number of reduction branches from a given statement into sub-statements and so on increase exponentially and possibly indefinitely i.e. the state-space of possible reductions from a given statement may be infinite. Thus, theorem proving is more often done with what is known as a proof-assistant, allowing for a combination of partial automation wherever possible and manual intervention wherever necessary. The degree of automation achievable in a proof-assistant is largely dependent on the user's ability to create and apply automation techniques. Another key factor in achieving automation is the form of the theorem to be proven. That is, some theorems lend themselves more easily to automation than others.

The remainder of this section is broken down as follows. In Section 2.5.1, tools relating to real-time systems are discussed. Hybrid tools are covered in Section 2.5.2. Theorem provers are the subject of Section 2.5.3.

[1] For information on how to use the tools, the best option is probably to visit the tool website. It is also instructive to download a tool and attempt modelling in order to really learn about the tool.

Then symbolic tools are visited in Section 2.5.4. Following this some general work on formal methods tools is explored in Section 2.5.5.

2.5.1 Real-time Tools

The seminal theory of Timed Automata (TA) [Alur and Dill, 1994] inspires the work of many existing real-time tools. These tools allow for the modelling of systems with a mixture of discrete behaviour and what can be called delay behaviour i.e. the passage of time. One of the most popular tools of this kind is UPPAAL, which allows Timed Automata to be built graphically by a user and then subjected to analysis in terms of LTL properties. That is, certain properties can be written about a UPPAAL model in LTL and these can be checked automatically by the state-space exploration algorithms of UPPAAL. A good introductory article briefing the reader on how to build and analyse models in UPPAAL is given in [Larsen et al., 1997]. Readers are also referred to "A Tutorial on UPPAAL" [Behrmann et al., 2004].

While the UPPAAL tool is still being developed and updated, there are more recent tools in this area. Real-Time Maude [Ölveczky and Meseguer, 2007] trades off decidability of its analysis techniques for expressiveness. That is, it allows more general properties to be specified and analysed than other tools with decidable algorithms, but the analysis is not guaranteed to find all behaviours of the system. The language is suited to the description of object oriented systems. A "very simple example" of a clock demonstrating the use of this tool is given in [lveczky and Meseguer, 2004].

Shrinktech [Sankur, 2013] is a tool concerned with the robustness analysis of automata. A downside to developing abstract models of real systems is that sometimes properties may be true only because of a theoretical nuance not present in the real world. Robustness analysis aims to filter out these "false positive" results, so to speak.

The opposite of a robust system might be called fragile or sensitive. For example, imagine a railroad crossing system in which a car will only enter the crossing if the time to cross is less than or equal to the time until the train arrives. Then if the two times are *exactly* equal, the car will, in theory, be able to enter the crossing and exit just at the instant that the train arrives. While in an abstract sense, this is safe behaviour, i.e. there were no collisions, it is clearly unrealistic. For example, the exact values of the crossing time and the arrival time of the train will in reality never be known. Therefore, a real system will factor some redundancy into its design e.g. the car will only cross if it has more than enough time to do so, where *more* is a factor of redundancy decided by a domain expert. Such a system can be said to be robust, i.e. capable of tolerating minor perturbations to its parameters while maintaining safe operation.

The question then is, how can such a notion of robustness become integrated into the abstract systems models? Recent work in this area [De Wulf et al., 2004] has begun to address this issue. Guards on the edges of an automata are enlarged in order to explore if any new undesirable behaviours become present. If no new behaviours are present within some enlargement of the guards then the automata is in some sense robust. The dual notion of this is guard reduction or shrinking, and the question asked here must be: does shrinking the guards cause the loss of any desirable behaviours? If not, then the automata is in a similar sense robust. The Shrinktech tool [Sankur, 2013] has been developed towards the analysis of automata under such shrinking perturbations.

Another recent real-time tool is PSyHCoS: "Parameter Synthesis for Hierarchical Concurrent Real-Time Systems" [André et al., 2013]. This allows for such systems to be modelled and then provides parameter synthesis and model checking over these systems. Parameter synthesis is the automatic generation of constraints upon abstract parameters in order to satisfy certain properties. That is, rather than specify exact parameter values, a user specifies them symbolically along with some property, and the tool automatically deduces constraints that must hold over the parameters in order for the property to be true.

2.5.2 Hybrid Systems Tools

Hybrid systems can be thought of as generalisations of real-time systems in that the continuous variables do not need to evolve at the same rate relative to each other. An array of tools have emerged in the past fifteen years for the modelling, analysis and verification of hybrid systems. Many of these tools face extreme challenges in practically implementing efficient algorithms for searching the state spaces of these systems. This is because not only do the state spaces grow exponentially with the size of models, but also because the shape of these state spaces can be non-trivial due to the possible existence of non-linear hybrid behaviours. Searching these non-linear state spaces requires techniques beyond those found in Timed Automata, whose state spaces are linear. The following list itemises a number of existing tools in the literature, approximately in the order that they emerged.

- Charon [Alur et al., 2000] allows for the modelling of hybrid systems composed in parallel and communicating via shared variables. In a similar fashion to many of the recently emerging network languages of Section 2.3.2, this language takes a two-tiered approach, separating the notions of agent and mode. Continuous update behaviour is modelled by either algebraic constraints, differential equations or invariants.

- CheckMate [Silva and Richeson, 2000] employs MATLAB as a back end towards the verification of hybrid systems. In order to allow the verification, the infinite state hybrid system is approximated to a finite state system called a quotient transition system.

- HyTech [Henzinger et al., 1997] is a popular tool for the modelling of hybrid systems. It supports linear hybrid systems. The state spaces of these systems are polyhedral i.e. their boundaries have straight edges. The tool provides model checking algorithms over such systems and can also be used for parametric analysis i.e. for computing constraints on parameters of a system for which some property holds. An extension of HyTech called HyperTech [Henzinger et al., 2000] improves upon the HyTech tool, allowing for the specification of non-linear dynamics. Another advantage of this tool over its predecessor is that the underlying algorithm no longer suffers from what is called “arithmetic overflow”, i.e. round-off errors causing some states to be missed in the state-space search. A review paper on HyTech [Henzinger et al., 2001] exposes some shortfalls of the tool. In particular, systems are quite limited in their scope due to the state-space explosion problem, or what is referred to in that paper as “the curse of dimensionality”.

- PHAVer a.k.a. Polyhedral Hybrid Automaton Verifier [Frehse, 2005] also exceeds the HyTech tool in terms of its abilities, particularly by eliminating arithmetic overflow.

- HySAT [Fränzle and Herde, 2006] combines linear programming state-space searching techniques with satisfiability (SAT) solver techniques. It uses bounded model checking to search the state-space of systems i.e. it looks ahead to a bounded number of states for a property. The claim made is that this tool outperforms its predecessors by exploiting search optimisations e.g. state-space pruning.

- KeYmaera [Platzer and Quesel, 2008] includes two languages. On the one hand, hybrid programs are abstract representations of hybrid systems. Differential dynamic logic on the other hand is used to specify correctness properties over hybrid programs. The KeYmaera tool allows for a mixture between automated techniques and manual proof techniques to prove the correctness of systems. At the back-end, KeYmaera is linked to a number of underlying solvers such as the Z3 theorem prover [de Moura and Bjrner, 2008].

- Breach [Donz, 2010] is a tool that uses numerical methods to simulate deterministic hybrid systems. It does not exhaustively check state spaces, but rather estimates infinite sets in a finite number of simulations. The tool is still at quite an early stage of development and does not seem to be overly user-friendly.

- HybridSAL [Tiwari, 2012] converts a specification of a hybrid system and a safety property into a discrete system and safety property that can then be verified by an existing discrete tool called SAL. The guarantee

of this conversion is that the correctness of the discrete system w.r.t. its safety property implies the correctness of the original hybrid system w.r.t. the original safety property. In order to convert from a continuous to a discrete system, a method called relational abstraction is employed: the continuous transition relation is over approximated into a discrete one. An advantage of this approach is that it breaks the burden of verification into two tasks: computing a good relational abstraction and verifying the abstraction using formal methods.

- Flow* [Chen et al., 2013] uses Taylor models, based on the Taylor polynomial series expansion, to generate what are called "flowpipes" of non-linear hybrid systems. A flowpipe is a "set of states reachable by continuous dynamics from an initial set within a given time interval" [Chen et al., 2012]. The advantage of this tool over many others is the support for non-linearity.

2.5.3 Theorem Provers and Proof Assistants

Another approach to tool supported formal methods is automated theorem proving or proof-assistant engines. Theorem provers are tools in which theorems can be specified and proofs automatically generated by underlying algorithms. Proof assistants on the other hand are tools in which theories can be encoded and theorems stated, but the proof burden falls ultimately upon the user. Proof assistants may incorporate some level of automation, making the line between them and fully automatic theorem provers unclear. A number of these tools are listed below.

- Coq [Huet et al., 2007] is a proof-assistant tool that uses a dependently typed functional specification language called Gallina. Functions in are treated as first class elements of this language. Also propositions can be specified, forming a special kind of type whose objects are proofs. Tactics allow for proof terms to be automatically built rather than manually specified by a programmer. A more detailed exposition of this tool its associated language appears in Chapter 7.

- The Isabelle theorem prover [Paulson, 1989] is based on higher order logic (HOL). A feature of this tool is "a function (called a tactic) mapping a goal to sub-goals" [Paulson, 1989]. A tactic is the inverse of an inference rule. That is, while an inference rule takes "simpler" components and combines them together to form a more complex component, a tactic does the opposite, reducing a goal statements to simpler statement and so on until eventually known results are reached.

- Simplify [Detlefs et al., 2005] is an automatic theorem prover for program checking. The tool is suited to software verification rather than the verification of hybrid/real-time or distributed systems.

- In [Brucker and Wolff, 2013], the Isabelle/HOL theorem prover is extended to equip it with the capability of test-case generation for software testing.

2.5.4 Symbolic and SAT/SMT Solvers

A number of the tools fall into the bracket of Satisfiability (SAT) solvers or Satisfiability Modulo Theories (SMT) solvers. SMT solvers are based on the pre-existing SAT solvers, which attempt to find solutions to formulas by effectively checking all the values of the variables therein. In [Barrett et al., 2009] SMT solvers are described as extensions to SAT solvers where a "background theory" is assumed to exist, constricting the state-space to be explored and thus speeding up the search process. A symbolic model checker is one in which the state-space is divided into groups or "symbolic states" and then these are checked all at once rather than each individual state be visited. The following is an account of some of the main tools of this kind.

Alloy is a state based formal modelling language with an associated SAT solver for verification purposes. Alloy models take relations as a primitive i.e. all structures are ultimately built using only relations. There is an object oriented style of specification in the language but it is just syntactic sugar for the underlying relational model. Alloy is interesting because the number of elements in each relation need not be specified. For this

reason, a model is quite abstract, but the cost of this is that models can only be checked up to some finite scope e.g. an upper bound must be provided to the SAT solver and the YES/NO result given is only valid up to that scope. Alloy does not directly support a model of real time and it is difficult to see how it could be encoded into the language. For an introductory text on the subject of Alloy, see [Jackson, 2006].

Z3 is an SMT solver intended for "software verification and software analysis" [de Moura and Bjrner, 2008] though it has been used for the specification and verification of a model similar to Comhordú [Asplund et al., 2012]. This qualifies as related work and is further explored in Section 8.4. The MathSAT 4 SMT solver is "explicitly designed for being used in a formal verification setting" [Bruttomesso et al., 2008]. It has been succeeded by MathSAT 5 with improvements supporting incremental solving, arrays and floating point numbers among other things.

Symbolic model checkers also belong in this genre. To name a few there is NuSMV [Cimatti et al., 2002] and SAL/SAL 2 [de Moura et al., 2004]. The former is used for analysis of synchronous finite state and infinite state systems. The latter is also referred to as the Symbolic Analysis Laboratory and is for the specification and analysis of state machines. Both use a mixture of binary decision diagrams (BDD) and SAT based methods in their analysis engines.

2.5.5 Other Tool Papers

The remainder of the tools cannot be categorised into any of the above sections, nor is there any obvious further sub-classification applicable to them, therefore they are collected together in this section and discussed in the following points.

- A number of model checkers use linear temporal logic (LTL) or a variant thereof as a language for specifying system properties. DiVine is "a tool for LTL model checking and reachability analysis of discrete distributed systems" [Barnat et al., 2010]. The Maude LTL model checker [Eker et al., 2004] allows states with "data types of infinite cardinality" e.g. reals. SPIN [Holzmann, 1997] is a model checker for distributed systems. It is geared more towards software systems rather than hardware, hybrid or real-time systems such as those related to Comhordú. The modelling language for SPIN is PROMELA, short for Process Meta Language. The verification language is LTL#, a variant of LTL.

- A synchronisation skeleton [Clarke and Emerson, 1982] is an abstract version of a program concerned only with synchronisation details i.e. steps that do not pertain to synchronisation are left out of the skeleton. In [Clarke and Emerson, 1982] a method for constructing a synchronisation skeleton for a program from a temporal logic specification is proposed. That paper is one of the earliest works in the field of model checking.

- The CVC [Stump et al., 2002] tool, standing for "Cooperating Validity Checker" combines decision procedures from arrays, inductive datatypes and linear real arithmetic in its underlying engine. Formulas in the specification language are either conjunctions or disjunctions of a mixture between equalities and inequalities over expressions.

- ICS [Fillitre et al., 2001] is shorthand for "Integrated Canonizer and Solver" which is a tool with a decision procedure over a functional language.

- MCMA [Lomuscio and Raimondi, 2006] is a "Model Checker of Mulit-agent Systems" extending temporal logic by adding to it additional modalities, allowing for the specification of epistemic, cooperative and correctness properties. The language of specification for this tool is the Interpreted Systems Programming Language (ISPL).

2.5.6 Tools: Discussion

This section examines tools in general in terms of their applicability to the problem at the heart of this thesis i.e. the formalisation and verification of Comhordú. Initially it seems as though a model checker or some related

form of automatic analysis tool would be well suited to the automatic verification of Comhordú. However a previous effort along these lines [Bhandal et al., 2011a] has uncovered a number of drawbacks to the tool approach; this is further explained in Section 3.5. Described in Section 3.5 is a partially successful but overly simplified formalisation of Comhordú in UPPAAL. This highlights the limitations of automated tools in general and provides good support for abandoning model checking in favour of partially automated theorem proving. The remainder of this section argues against the use of tools such as model checkers in general terms.

To begin with, many tools are intended for the modelling and verification of software rather than hardware or indeed more rarely hybrid systems. That is, they generally aim to model systems with "only countably many distinct behaviors" [Holzmann, 1997] in contrast to the infinitude of behaviours integral to a continuous model such as Comhordú.

Those that are hybrid are generally limited e.g. "systems with complex relationships between multiple parameters and timing constants can quickly lead to arithmetic overflow" [Henzinger et al., 2001]. This is partially because hybrid systems is quite a recent field but also because hybrid systems, with their potentially non-linear continuous behaviours, are inherently complex. At the cutting edge of hybrid systems research is the KeyMaera tool, which seems to be the most flexible tool for modelling hybrid systems. However, this tool seems more suited to the specification of reasonably complex continuous dynamics via differential equations, whereas the exact dynamics of entities in Comhordú are not a consideration of this work. Nonetheless, KeYmaera may be useful in future explorations of Comhordú, particularly specific instances with particular dynamics. This is covered further in Section 8.5.

Orthogonal to the difficulties faced by hybrid tools are the complications introduced by concurrency. According to [Holzmann, 1997], "[c]oncurrent systems is, compared to civil engineering or physics, a relatively young discipline and it is not surprising that comparable tools are still somewhat scarce." While this quote dates back to 1997, it still highlights the the relative immaturity of the field with respect to other engineering disciplines that are hundreds of years old. This of course applies to computer science in general.

Coupled with this immaturity is a genuine complexity introduced by concurrency that seems inescapable. In most models of concurrency, as the number of concurrent processes in a system increases, its number of states tends to increase exponentially. This is known as the state-space explosion problem. The resulting space/time demands upon any real machine hoping to run a model checking algorithm "are an often neglected issue in formal verification" [Holzmann, 1997]. Nonetheless, all tools thus far encountered suffer from this problem. While there is an increasing number of methods available to tools towards the reduction of the complexity of searching these rapidly growing state spaces, it does not seem possible that this complexity will ever be outright eliminated. Thus even if concrete instances of Comhordú were to be verified, there would be a limit on their size due to this exponential increase in model checking time.

Furthermore, it is difficult to imagine how Comhordú could be specified using any of the tools mentioned, since it is specified in terms of symbolic parameters whereas "[model checkers] need concrete numbers for most parameters" [Brucker and Wolff, 2013]. Also systems can generally be verified only for a fixed number of components: "When the number of nodes is fixed a priori, formal models... can be verified by using finite-state model checking" [Delzanno et al., 2010]. On the contrary, Comhordú is specified in terms of abstract parameters, uninterpreted functions and an unknown number of entities. Certainly, it would be possible to instantiate some or all of the parameters of Comhordú and fix the number of entities involved. Furthermore, any complex behaviours of the system could be "abstracted away" in order to make it "fit" into a particular model checker. Indeed, this is exactly the approach taken in the work presented in Section 3.5. As explained in that section, while this abstract-and-simplify approach does have its merits, it is not detailed enough to allow any conclusions to be drawn that the original specification is indeed correct.

In summary, the following list outlines the pros and cons of model checkers, theorem provers and SAT/SMT solvers in terms of their applicability to the modelling of Comhordú. Further attention is given to these in Table 2.1, which provides an evaluation of the various formal methods considered in this chapter.

- Model checkers are useful in that they provide fully automatic analysis. However, they are constrained

in terms of the systems that can be analysed. On the one hand, state-space explosion means that the time to analyse systems in general grows exponentially with the size of its specification. A further constraint is that system parameters usually need to be concrete if they are to be subjected to analysis/verification procedures. For example, the number of components in a system is usually fixed prior to its specification.

- Theorem provers and proof assistants are beneficial in that they allow a user to encode any theory whatsoever. Another advantage of these tools is the potential for exploitation of existing theories. The downside though is that full automation is generally lacking for this type of tool.

- The satisfiability (SAT) problem is known to be NP-complete. As such, SAT solvers suffer from similar complexity problems to model checkers. While SMT solvers aim towards reducing the inherent complexity of the SAT problem by introducing a background theory, "it is infeasible to build a procedure that can solve arbitrary SMT problems" [De Moura and Bjørner, 2009]. That is, the optimisations of SMT rely on the existence of an established background theory. There is no such theory for the Comhordú model, so any attempt to exploit SMT would need to bridge the gap between existing theory and the specifics of Comhordú. It is unclear at this point how exactly this could be achieved.

2.6 Discussion of Literature

Recall the goal of this thesis is to formalise and verify a coordination protocol called Comhordú. The setting for Comhordú is the field of distributed systems covered in Section 2.1 and the more specific area of Future Cities and Vehicular Computing (FCVC) from Section 2.2. An argument in favour of the formalisation of Comhordú is justified by general arguments towards the formalisation of systems that appear at the beginning of Section 2.3. With this motivation in mind, various formal methods are explored in Section 2.3, Section 2.4 and Section 2.5 and their relevance to the problem at hand is discussed. Let us briefly recap on this material in the following.

Coordination pertains to distributed systems in which many entities operate concurrently. Such systems can consist of cars, unmanned aerial vehicles (UAV), robots, or underwater vehicles, to name a few widespread examples. Comhordú is a coordination protocol for systems of mobile entities communicating over a wireless network. The means by which Comhordú achieves coordination differs from the majority of its predecessors in that it does not rely on consensus. Rather, Comhordú defines a notion called responsibility, whereby each entity ensures safety relative to itself and so system-wide safety emerges. At the time of its original conception, Comhordú was intended for application in the domain of Future Cities and Vehicular Computing (FCVC) research. However, is seems that due to its generality, Comhordú is more suited to less-structured applications like UAV swarms, whereas domain-specific solutions are more applicable in FCVC scenarios.

There are a number of arguments in favour of the use of formal methods in general. These are important in establishing motivation for the formalisation of Comhordú. The direct benefit of specifying a system in a precise language is that there can no longer be any ambiguities or uncertainties in the specification. A designer is forced to address any design issues that arise at the specification stage, avoiding potentially costly consequences if these issues manifest themselves at a later stage in the design process. Coupled with this is the possibility of verification and simulation. Formal specifications can be analysed in terms of their behaviours, either by hand or with the aid of machines. This can constitute either explorations of various behaviours or exhaustive checking to ensure no "bad" behaviours occur. Alternatively, conditions can be stated in relation to such specifications and formally proved, possibly with the aid of a proof-assistant or theorem-prover.

Formal methods constitute a constellation of methodologies, languages and tools all geared towards the precise specification and analysis of systems. In the interest of formalising Comhordú, various formal methods are investigated. Formal specifications languages are analysed in Section 2.3 in terms of their applicability for specifying real-time systems with wireless communication and spatial mobility. Real-time formalisms are inspected in greater detail in Section 2.4. While they are deemed unsuitable for specifying Comhordú, it is

mentioned that some of the ideas from these formalisms are employed in the final proof of safety for Comhordú. Tools are attractive to any systems designer in that they offer the advantage of automatic analysis. A number of tools are listed in Section 2.5. Unfortunately, tools for fully automatic analysis seem to be too limited in their expressiveness in order to formalise and verify Comhordú.

We conclude this section with a rather broad overview of the formalisms explored so far. Table 2.1 is an evaluation of various types of formalism in terms of a number of attributes that are important to the Comhordú model. Neither the list of attributes is necessarily complete, nor is the information in the table based on any sort of exact science. The idea is just to provide a rough summary of the various types of formalisms encountered so far and highlight their strengths and weaknesses relative to each other in general. The table is by no means a substitution for the more detailed descriptions of the individual formalisms found in the previous sections.

The attributes covered in the table are as follows.

Real-time	Is the language suitable for the specification of real-time systems?
Mobility	Is the specification of systems of mobile entities supported?
Distributed	Does this type of formalism support the specification of systems of entities running concurrently?
Abstract	Does this type of formalism support symbolically represented parameters and an arbitrary number of components?
Auto	Are there automatic analysis/verification algorithms available, possibly as part of a tool?

For each attribute of a particular type of formalism, an evaluation is given. The evaluations are listed below.

NO	The attribute is generally not supported by this type of formalism.
RARE	The attribute features in few formalisms of this type, but is absent from most.
SOME	Some formalisms of this kind support the attribute.
MOST	Most of the time the attribute is supported, but there are some formalisms of this kind that do not support it.
YES	The attribute is in general supported.
LIMITED	The attribute is supported in most cases but there are limitations to the amount of support provided. For example, in hybrid systems, automatic analysis is supported but the range of systems analysable is quite small due to the practical impediment of the state-space explosion problem.
POSSIBLE	It is possible to support this attribute but this will require work on the part of the user. For example, a theorem prover can generally support most theories, but these must be specified by the user, or written by somebody else and imported i.e. as an external library.

Table 2.1: Evaluation of formalisms based on various attributes.

Formalism	Real-time	Mobility	Distributed	Abstract	Auto
Classical Process Algebra	NO	NO	YES	YES	SOME
Network-based Languages	RARE	SOME	YES	YES	RARE
Hybrid Systems	YES	YES	RARE	RARE	LIMITED
Temporal Logic	SOME	RARE	RARE	SOME	SOME
Real-time Languages	YES	RARE	SOME	MOST	SOME
Real-time Tools	YES	RARE	SOME	RARE	LIMITED
Model Checkers	SOME	RARE	SOME	RARE	MOST
Theorem Provers	POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE	LIMITED
Proof Assistants	POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE
SAT/SMT Solvers	POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE	LIMITED

Clearly from the table, proof-assistants are the most general type of formalism, allowing for the specification of whatever theory the user decides to write. However the price to pay for this generality is that a user will in most cases have to spend a great deal of time building up foundational theories. Also for automatic analysis and verification support a user will generally need to write custom-built tactics. The approach taken in this thesis is to use a proof-assistant called Coq to encode a modelling language and model of Comhordú. Wherever possible, pre-existing theories are imported. Also, the modelling language builds upon other formalisms, particularly process algebras.

Chapter 3

Towards a Formal Coordination Model

The original presentation of Comhordú concludes with a proposition for formalisation: "A significant extension to this work would be to provide a formal proof of Comhordú, by analytically demonstrating that the requirements derived for each contract type are sufficient to ensure the safety constraint" [Bouroche, 2007]. This chapter begins with a summary of the original Comhordú model and then investigates various avenues towards its formalisation. This investigation highlights some of the challenges associated with formalisation. To address these challenges, various design choices are made, some of which entail compromises to the original goal of formalisation. The resulting compromises are assessed and the chapter concludes by outlining the current choice of methodology.

The section breakdown of this chapter is as follows. Section 3.1 gives an abbreviated account of the original Comhordú model. An assessment of this model in terms of its amenability to formalisation follows in Section 3.2. Revisions to the Comhordú model towards the facilitation of its formalisation are put forth in Section 3.3. These revisions and their implications are assessed in Section 3.4. A previous formalisation effort using the modelling and verification tool UPPAAL is summarised in Section 3.5. This highlights the limitations of a tool-based approach and leads to the currently adopted approach, which is outlined in Section 3.6.

3.1 Original Comhordú Model

Recall the motivations for coordination from Section 1.1. To recap, coordination is defined as "the management of interactions both amongst entities, and between entities and their environment, towards the production of a result" [Bouroche, 2007]. A coordination model in this context is a theoretical framework which facilitates the development of schemes for the coordination of systems of entities. Included in such a model will be a characterisation of the behaviour of these systems e.g. message sending between entities or movement of entities.

Comhordú is a coordination model developed in [Bouroche, 2007] which incorporates a novel approach to coordination based on a notion called responsibility. The idea is that certain entities in a system are elected as responsible and these entities must then independently ensure correct operation of the system without necessarily being granted the consent of other entities. This contrasts with previous approaches to coordination based on the notion of consensus e.g. [Nett and Schemmer, 2004, Julien and Roman, 2004], whereby coordination is achieved by all entities agreeing on a certain plan of action. The responsibility approach is particularly suited to scenarios in which network coverage and sensor information are unreliable because in these scenarios it is sometimes impossible to communicate with a sufficient neighbourhood of other entities in order to realise consensus. Included in Comhordú is a model of unreliable communication and the environment, a number of schemes for coordination based on the notion of responsibility and a methodology for deriving constraints on entity behaviours in specific scenarios in order to ensure safe behaviour of entities.

The following sub-sections provide a synopsis of the Comhordú model as it appears in [Bouroche, 2007].

Section 3.1.1 establishes the main features Comhordú. The key idea of responsibility is discussed in Section 3.1.2. Section 3.1.3 summarises a methodology of Comhordú by which behaviours are derived for entities so that they ensure system safety. Section 3.1.4 describes a tool used to implement the Comhordú methodology. The account of Comhordú given here is succinct. For the full presentation the reader is referred to the original thesis [Bouroche, 2007].

3.1.1 Comhordú Preliminaries

In this section, the main concepts underlying the Comhordú model are briefly outlined. These concepts constitute a basic framework within which coordination schemes can be proposed. The following itemises what are believed to be the fundamental features of the Comhordú model.

- An environment in Comhordú consists of a collection of elements. Elements can be distinguished as being either passive or programmable. The latter are referred to as entities. The entities are "physical mobile entities, not software agents" [Bouroche, 2007]. Entities can be sub-divided into a number of types e.g. cars, trucks, emergency vehicles etc.

- Comhordú incorporates models of the media by which entities communicate i.e. communication models. These models are considered as "hypotheses upon which Comhordú builds" [Bouroche, 2007]. That is they are specified in terms of a number of high-level conditions that are assumed to hold true in an environment. There are two communications models. Direct communication pertains to the sending of messages across a wireless medium. The model of direct communication is called the Space-Elastic model (SEM). Indirect communication involves entities using sensors and actuators to sense and change their environment respectively. For example, a car may alert other cars of its presence by sounding its horn.

  A further explanation of SEM is warranted here, since SEM is a key element of Comhordú. Included in this model is the notion of coverage. The coverage about an entity is the area to which it can deliver timely messages. That is, the coverage is the area within which an entity can deliver messages within a predetermined time bound called $msgLatency$. With time, this area may change, hence the term "space-elastic". The key guarantee of the Space-Elastic Model is bounded notification of coverage to entities. That is, entities are notified within a predetermined time bound, called $adaptNotif$, of the area to which they can communicate. The implementation details of SEM are not considered as part of Comhordú. Nor are failures permitted: "Our work assumes the Space-Elastic Model but does not tolerate any failures of the Space-Elastic Model" [Bouroche, 2007].

- At any time, entities are assumed to have a state. The state of an entity can be decomposed into a collection of possible state variables. Examples of state variables are position and velocity.

- A scenario is a collection of entities along with a priority list and a safety constraint. These last two concepts are addressed later. A scenario is said to be solvable if there exists some set of requirements on entity behaviour such that adherence to these requirements by entities implies safety in the scenario. These requirements are then called the solution to the scenario. In a certain sense, the Comhordú coordination model can be viewed as a theory/framework for developing scenarios that are safe.

- Safety is expressible in the Comhordú model via a safety constraint language. A safety constraint expresses a condition that should always be true in a certain scenario. Safety constraints in Comhordú are defined in terms of what are called incompatibilities. There are four types of incompatibility which can be classified as follows.

  - Comparison between a state variable and a value e.g. $speed < 14$.
  - Comparison between two state variables e.g. $speed_1 = speed_2$.

  - Comparison of the distance between two positions with a value e.g. $d > 5$ where d is the distance between two entities.
  - Condition on the cardinality of a set of entities satisfying some condition on their state variables e.g. "there are at most 7 entities whose speed is greater than 5".

- Modes are abstractions of the state of an entity. Different mode abstractions can be made for a system depending on the notion of safety associated with that system. For every entity type there is a set of modes and a mode transition diagram which is a directed graph over these modes. This transition diagram captures which modes are reachable from other modes. A set of modes is said to be compatible with respect to a safety constraint if any collection of entities in those modes is safe according to the safety constraint, regardless of the values of the other state variables. Fail-safe modes are distinguished modes that are always compatible with all other modes.

- A priority list of modes for a particular scenario captures the fact that some modes are more important than others. The idea is that entities in a particular scenario should behave according to the priority list in question. That is an entity should always choose the highest priority mode available to it.

- Contracts are specified as "conditions for safely performing certain actions" [Bouroche, 2007]. Entities in particular scenarios can be assigned contracts, the adherence to which should entail safety. Contracts are revisited again in more detail when responsibility is explained in Section 3.1.2.

- A goal is a condition on state variables that an entity is trying to achieve. It is desirable that entities in a scenario should progress towards their goals. For example, a car may be trying to reach a certain position.

3.1.2 Responsibility: an Alternative approach to Coordination

Central to Comhordú is the notion of responsibility as a strategy for coordination. This strategy is applicable in real-time systems with unreliable coverage for which, it is argued in [Bouroche, 2007], a consensus-based approach is unsuitable. Responsibility addresses the difficulties of coverage degradation by exploiting the coverage feedback of SEM and from this information deducing when it is safe to act. As pointed out in the original development of Comhordú, responsibility only "caters for a class of applications for which some entities can, independently of other entities, take some actions to ensure that the safety constraints will not be violated" [Bouroche, 2007]. Another property of responsibility is that it is potentially over-conservative. That is, the number of responsible entities may exceed the minimum number sufficient to ensure coordination. This is because entities are acting independently and are thus in general unaware of responsible neighbours.

The concept of responsibility is embodied by the idea of contracts, which are briefly mentioned in Section 3.1.1. To reiterate somewhat, contracts are conditions to which entities must adhere in a particular scenario in order to ensure safe behaviour in that scenario. Different entities in a scenario can be assigned different contracts. Certain entities are elected as "responsible". Every responsible entity in a scenario must adhere to its contract in order to ensure safety in that scenario. The set of responsible entities is not fixed, however. That is, over the course of time, an entity can transfer responsibility from itself to other entities. This transfer of responsibility is achieved via message sending and can thus only take place when coverage is in some sense sufficient.

The basic building-blocks of contracts are called coordination primitives. These are basic actions that entities can take in order to maintain safety. There are three types of coordination primitive, as outlined below.

- Adapting entails an entity changing its behaviour to cope with a changing environment.

- Delaying involves postponing a planned change of action until it is safe to do so.

- Transferring responsibility amounts to warning other entities of current/planned actions so that they become responsible for adapting accordingly. This is only possible if communication is sufficient. Otherwise responsibility must be upheld via the adaptation and delay primitives.

Building upon the above primitives, three types of contract can be defined. The following list explains these contract types. Each successive contract type inherits all the features of its predecessors. Thus only new features are described.

A contract without transfer enforces an entity to take full responsibility for safety based on sensor information alone. The entity must remain responsible indefinitely according to this contract.

A contract with transfer without feedback allows an entity to send messages to other entities, transferring responsibility. When communication degrades the entity under contract must take full responsibility as in the case of a contract without transfer. With this contract, a responsible entity must send messages to neighbouring entities when in a non-fail-safe mode (NFSM). It also must warn other entities when entering a NFSM. The requirement upon other entities is to adapt their behaviour accordingly upon receipt of a message from a responsible entity.

A contract with transfer and feedback augments the previous contract in that entities that send messages can receive feedback from other entities. This generally entails longer wait times before progress can be made because an entity must wait for its message to be delivered and then it must wait for replies from surrounding entities.

Given a particular scenario, involving a collection of entities and a safety constraint, various contracts can be devised for the different constituent entities. The means by which this is done is examined in Section 3.1.3.

3.1.3 Methodology of Comhordú

A methodology is included as part of the Comhordú model by which contracts can be constructed for a particular scenario. Recall that a scenario includes a collection of entities and a safety constraint. A scenario is solvable it there exist contracts that can be assigned to each entity in the scenario such that adherence to contracts by all entities entails safety. The assignment of contracts along with the choice of responsible entities constitutes what is called a solution to a given scenario.

The methodology formulated in [Bouroche, 2007] filters valid solutions from the set of all possible combinations of contract types and choices of responsible entities for some scenario. The filtering is achieved by pruning this set, excluding all combinations that do not constitute solutions. Further filtering removes solutions that do not allow entities in the scenario to progress. A final stage in this process allows solutions to be ranked according to which ones admit the highest priority behaviours. A tool exists for automating this methodology. A number of scenarios are modelled with this tool in [Bouroche, 2007] e.g. a pedestrian traffic light and a system involving cars and emergency vehicles.

3.1.4 The ComhMod Tool

ComhMod is a tool for implementing the methodology previously outlined. Specifically, ComhMod is described in the original thesis as a means of facilitating "the development of applications composed of autonomous mobile entities" [Bouroche, 2007]. It supports a series of stages in this development, briefly described in the following.

Entity definition: Entities are defined with state variables of type Integer, Double, Boolean and Position (a pair of doubles). Variables of a user-defined type can also be added; these are represented as strings. Mode invariants are defined as conditions on state variables that must hold in a certain mode. A mode transition diagram is specified, defining which modes can transition to others.

Safety constraint specification: A safety constraint is specified for the scenario according to the safety constraint language of Comhordú.

Mode compatiability evaluation: Some algorithms are run to asses that incompatibilities do not happen in certain modes e.g. a mode might be associated with the constraint $speed < 10$, which ensures that $speed \geq 10$ never happens. A mode compatibility matrix is specified which gives, for any two entity types, a true/false value for every pair of modes. True means those modes are always compatible. Recall that compatible modes always respect the safety constraint.

Responsibility and contract type attribution: A set of solutions is generated. The user can then choose any solution from this set. Recall that a solution is a choice of responsible entities and contract types for all entities.

Requirements derivation: Requirements are derived for entity behaviours based on the solutions generated in the previous step.

Addition of numerical values: Specific values are assigned to the various parameters e.g. $adaptNotif$ and $period$.

Output: The output of this process is what is called a *sentient object skeleton*, which is an XML document that can be interpreted by a tool called MoCoA, which is briefly discussed in the following.

MoCoA is a middle-ware predating ComhMod for the description of *sentient objects*. It allows *sentient object skeletons*, to be interpreted; recall that this is the output format of ComhMod. In short *sentient objects* form a programming abstraction and are defined as follows: "mobile intelligent entities, that extract, interpret and use context information obtained from sensors, other sentient objects, and their computational infrastructure, to drive their behaviour" [Bouroche, 2007]. The MoCoA tool chain allows for sentient objects to be built and simulated. Simulation allows behaviours of the objects to be observed.

It must be emphasised here that ComhMod is not a formal model. Nor is there any proof given that it is correct in terms of guaranteeing safety. Rather, it allows individual scenarios with specific parameters and safety constraints to be built and tested. The limitations of this are:

- Parameters must be given numerical values.
- Only one scenario at a time can be tested.
- For a particular scenario, a conclusive guarantee that the solution guarantees safety is not given. Rather, a number of specific behaviours can be simulated, and these alone can be shown to be safe.

3.2 Assessment of Comhordú

The objective of this thesis is to address the task of formalising Comhordú. To reiterate, the original proposition is to "provide a formal proof of Comhordú, by analytically demonstrating that the requirements derived for each contract type are sufficient to ensure the safety constraint" [Bouroche, 2007]. The feasibility of realising this goal is now assessed. In particular, this section reflects on the process of analysing Comhordú and attempting to arrive at an understanding of the model necessary to facilitate its formalisation. To begin with, in Section 3.2.1 inherent ambiguities in the aforementioned statement are identified and addressed, culminating in the formulation of a more precise, detailed objective. Potential problems with the realisation of this objective are then raised in Section 3.2.2.

3.2.1 Disambiguation of Goal Statement

The first question to ask is, what exactly is desired via formalisation? To restate the aforementioned goal in a slightly different manner, formalisation should address whether or not adherence to the derived contracts by entities guarantees safety in every solvable scenario. While this seems like a perfectly reasonable statement, it is still rather ambiguous from a formal point of view. Recall that a scenario is a collection of entities and a safety constraint. A safety constraint in turn is a condition which can be interpreted over a set of entities in certain states. A contract, on the other hand, is a set of requirements on entity behaviour. Hence it does not make sense to reason about contracts without a model of behaviour. The following points embody the process of reasoning towards the introduction of a behavioural model with the ultimate aim of formulating a more precise goal for formalisation. The ensuing discussion is intentionally informal and written in an exploratory style.

- To begin with, it must be asked should entities in a scenario have fixed states i.e. fixed values for all their state variables? Intuitively the answer to this is no. The entities themselves can be thought of as fixed components of a scenario while their states should be capable of change in some sense.

- The notion of change immediately suggests the need for a model of time; arguably change and time are one and the same thing. It is then plausible to assume that given a certain scenario, and perhaps an initial assignment of states to all of its constituent entities, the evolution of this scenario could be modelled as a function mapping each point in time to a collection of states for all the entities. The initial assignment of states would correspond to the value of the function at time 0. For example, at time 13 a car might be accelerating at speed 7 and an ambulance might be braking before reaching a traffic light, while another car might be switching from lane A to lane B; at time 14 the first car might be at speed 7.4, the ambulance might be stopped in front of the traffic light and the other car might have moved fully into lane B.

- It seems intuitive that “freezing” a scenario at a point in time would yield what might be referred to as a system state or merely a state of that scenario. The state would be composed of all the individual states of the constituent entities. The set of all possible states could be referred to as the state-space of the scenario.

- This model of time is often referred to as linear or deterministic. It seems to resonate with our intuitive understanding of time moving forward in a straight line as events occur along this line. However, a deterministic model of time seems too restrictive here. This is because in any abstract model, precision is necessarily lost with respect to all of the information in the real world e.g. the position and velocity of every single particle in space. However, often the omitted details in reality have some effect on the outcome of a system, rendering such an abstract model imprecise in some sense. This imprecision can be compensated by a non-deterministic time model. That is, rather than time progressing in a linear fashion, there might be many different futures to a given state.

- Amalgamating the previous points then, a scenario could be augmented by adding to it a behavioural model including a state-space and a potentially non-deterministic model of time over that state-space.

- A single behaviour of the system could be examined by tracing out a path, in some sense, through the state-space. Such a behaviour might be referred to as an execution or a run. Alone, this run would constitute a deterministic time model i.e. there would be a unique state of the system for every point in time relative to the run. Viewed another way, a run would be just one of an assortment behaviours allowable in the non-deterministic model.

- With a model of behaviour such as a state-space with runs, contracts could be interpreted. First of all, let us refer to an assignment of contracts to entities simply as an assignment. A run could then be said to adhere to an assignment if for every entity the conditions of its contract hold across the entire run. The

notion of a contract holding across a run is in itself somewhat ambiguous, though it is not given further elaboration here. Rather, the issue of contract adherence is addressed in Section 3.2.2.

Remark 3.1 (Simulations of ComhMod: Not a Model of Behaviour)**.** It may be observed that the simulations of ComhMod, mentioned briefly in Section 3.1.4, are demonstrations of entity behaviour. However, simulations do not constitute an explicit behavioural model. Rather, their behaviours are entangled with the entire specification of ComhMod, the underlying MoCoA tool chain, and the programming language in which they are written. ▲

Bearing all the above reasoning in mind, a more detailed, precise objective can be formulated for the formalisation of Comhordú. In particular, formalisation should address the truth/falsehood of the following: *Suppose there is a scenario the entities of which have been assigned contracts via the Comhordú methodology. Then for every corresponding run adhering to this assignment, all of its states are safe i.e. the safety constraint holds for each state in the run.* However, this is still not a precise statement in formal terms and requires further disambiguation in order for formalisation to progress. In particular runs are not precisely defined, and the notion of run adherence is unclear. Prior to establishing these concepts precisely, several potential challenges posed by the prospect of formalisation are addressed in Section 3.2.2.

3.2.2 Potential Problems with Formalising Comhordú

A number of potential problems arise in relation to the objective of formalising Comhordú. What are believed to be the main concerns in this setting are outlined below.

- To begin with, there are many concepts modelled by Comhordú, as can be seen from Section 3.1. None of these concepts in the original model are given formal definitions, i.e. a formal syntax and semantics. This leaves them somewhat open to different interpretations. Often the subtleties uncovered by considering these various interpretations lead to a large amount of analysis and detail, as can be seen from the previous dissection of the meaning of behaviour in Section 3.2.1. Seemingly simple concepts in the prose description of the model often become rather involved when rigorous definition is attempted. Thus the inclusion of all details from the original model seems intractable in terms of the size of the formal model this would generate. A large model not only takes time to specify but becomes difficult to understand and even more difficult to analyse/verify formally.

- The safety constraint language seems particularly problematic. While the presentation of this language in [Bouroche, 2007] is semi-formal in that it is defined using an inductive syntax, the underlying components upon which the language is built are somewhat ambiguous. For example the language allows for the comparison of state variables via relational operators. The exact meanings of "state variable" and "relational operator" are not given.

- Central to Comhordú is the notion of responsibility. However the meaning of responsibility is somewhat unclear from its treatment in [Bouroche, 2007]. On the one hand, the defining characteristic of responsibility is that the responsible entities alone can guarantee system safety by their correct actions. However, in the definition of a contract with transfer, a condition is placed on "other" entities that they must react to incoming messages. This seems to imply that these "other" entities are, in some sense, also responsible. For this reason, a simplification is suggested here that *all* entities in a system are responsible, with responsibility meaning that an entity obeys its contract, regardless of how simple that contract is e.g. reacting to incoming messages. This means that a solution to a scenario, rather than consisting of an assignment of contracts *and* responsible entities can be simplified to just an assignment of contracts. Every entity is then responsible to fulfill its contract.

- The notion of a contract holding for an entity over a run is difficult to define. A naive attempt at such a definition is to say that a contract holds for an entity and a run exactly when it holds, in some other sense, for every state along that run. A contract holding for a state intuitively would the be easier to define because states are simpler structures than runs.

However, it soon becomes clear that this concept of contract adherence is too simple. Contracts implicitly rely not just on the current state, but also on what has come before i.e. the history of that state. For example, a contract with transfer stipulates that an entity should broadcast periodically to transfer its responsibility. This periodicity of broadcast cannot be defined in terms of a single state alone, but instead relies on the time between successive broadcasts, hence spanning over a number of states.

- In order to develop a formal model, there needs to be some unifying theory. However, the breadth of possibility introduced by the generality inherent in such concepts as "state variable" and "entity type" means that different scenarios can potentially have vastly different behaviours. For example, a state variable could be a temperature, the image of an on-board video camera, the roughness of the ground surface etc. Similarly, an entity type might be ambulance, car, robot etc. Certainly leaving these concepts uninterpreted is advantageous in terms of the breadth of systems expressible. However, the development of a general model of behaviour for scenarios becomes difficult in this setting. The only behavioural elements that can be included in such a model are those that are common to all scenarios, and so the specifics of state variable evolution cannot be included in the model. To compensate for this, a degree of non-determinism can be introduced into a behavioural model.

  Alternatively, scenario specific behaviour can be modelled. For example, along the lines of the scenario appearing in [Bouroche, 2007], a scenario with an ambulance, a car and a traffic light could be modelled incorporating separate behaviours for each entity. The car and ambulance might be capable of moving between different speed ranges, changing lanes, stopping at the traffic light etc. The traffic light might be given the behaviour of periodically changing its colour from red to green to orange, maintaining each colour for a predetermined amount of time. However, this behavioural model would be specific to the scenario in question and new behavioural models would need to be formulated for other scenarios. This does not seem like a satisfactory option in light of the formalisation objective, which should show that contract assignments are correct in some sense for *all* solvable scenarios.

Addressing some of these potential problems, a revised informal coordination model is formulated. This is presented in Section 3.3.

3.3 Revised Comhordú Model

In light of the assessments made in Section 3.2, in particular the problematic aspects of formalising Comhordú highlighted in Section 3.2.2, a number of revisions are made here to the original model, rendering a new model. It is emphasised that the revised model is a key contribution of this work; as can be seen from Section 3.2, there is a significant amount of reasoning underlying the revisions that are made to the model here.

Remark 3.2 (Terminology: Comhordú vs. Revised Comhordú)**.** By a slight abuse of terminology, in the remainder of the thesis this revised model is sometimes referred to as simply the Comhordú model or just Comhordú, whereas the model presented in [Bouroche, 2007] is referred to as the original Comhordú model or just the original model. The remaining chapters of this thesis are based on the revised model. ▲

Overall, the revised model strips back a lot of detail from the original. The necessity in removing detail stems from the previously discussed issue of model size: a specification can become problematic if it grows too large, particularly if it is to be subjected to formal analysis/verification. Moreover, simplifying the model leads to a unified framework with a general characterisation of behaviour. That is, there is a single framework encompassing the entire model rather than a fragmented series of definitions of its various components. The alternative approach would be to specify individual behaviours for specific scenarios, but this does not seem satisfactory relative to the objective of this thesis, which is to prove Comhordú correct in general.

This section is further sub-divided as follows. Table 3.1 outlines a number of basic underlying components of the model. Section 3.3.1 presents a modified version of the Space-Elastic Model. An informal behavioural

model for systems of entities is developed in Section 3.3.2. In Section 3.3.3 a revised notion of safety is defined. In Section 3.3.4 a revised contract is devised, embodying the notion of responsibility.

3.3.1 A Modified Space-Elastic Model

To recap, the key feature of the Space-Elastic Model from the original Comhordú model is the provision of feedback to an entity of the state of its coverage within a predetermined time bound called $adaptNotif$. Coverage itself is defined as the area to which an entity can deliver timely messages i.e. the area to which messages can be delivered within a time bound $msgLatency$. This coverage feedback can be used by an entity to determine the areas to which its past messages have been delivered.

It becomes apparent upon analysis of the various types of contract in the original Comhordú model that the only use of coverage is to determine the area to which a previously sent message is guaranteed to have reached. That is, the coverage of an entity at a time when no messages are being delivered is redundant to the model. In light of this, there is no need for *continuous* feedback to the entity of its coverage. Rather, an entity only needs to be informed of the coverage of each of the messages that it sends.

The simplified Space-Elastic Model then provides to an entity, exactly $adaptNotif$ time units after it has delivered a message, the coverage to which that message was sent. For further simplicity, the coverage is assumed to be circular and so it is returned to the sender as a single number representing the radius of the circle within which other entities are guaranteed to have received the entity's message. Note that the centre of this circle is the position of the sender at the time the message was delivered.

To summarise then, this revised Space-Elastic Model captures message delivery and coverage notification in the following way:

- An entity can decide to initiate the sending of a message at any time t. The message is said to be sent at this time.
- This message is then guaranteed to be delivered to some radius r at time $t' = t + msgLatency$. The size of the radius is arbitrary, modelling the variability of coverage. All entities within the circle defined by the position of the sending entity at the time of delivery t' are guaranteed to have received the message. The message is said to be delivered at time t'.
- At time $t'' = t' + adaptNotif$, the sending entity is notified of the radius r to which its message was delivered. For the sake of precision, we say that the notification contains both the delivery radius and the content of the message delivered. This is to allow for the possibility of two different messages being sent at the same time by an entity but delivered to different coverages.

3.3.2 Behavioural Model

The initial Comhordú model does not include any explicit model of behaviour. This is probably because it is difficult to give a behavioural model without getting too formal. However, without any explicit behavioural component, the model is ambiguous. For example it is not clear whether time is supposed to be linear or branching. That is, given a scenario, is there only one possible sequence of states characterising the behaviour of that scenario or are there many possibilities? The latter seems to be a more accommodating interpretation i.e. it compensates for the existence of unknown variables. To alleviate this ambiguity an explicit behavioural model is sketched here. The behavioural model given here is not precise, but rather constitutes a template or schema upon which a formal model of behaviour can be defined.

To begin with, an entity is some physical object capable of moving about in space. The class of entities of interest here are those that are autonomous, controlled by on-board software; the original model defines entities as "autonomous mobile computer systems" [Bouroche, 2007]. An entity is assumed to have a number of state variables. In particular, it is assumed to have a position and a mode. An entity is said to be in a particular

Table 3.1: Basic Underlying Components and Assumptions of the Revised Comhordú Model.

Types	
$\mathbb{M}$	The set of all modes.
$\mathbb{T}$	A set modelling the time domain. In this model the real numbers are chosen for the time domain, though any dense time domain would suffice.
$\mathbb{L}$	Two dimensional Cartesian coordinates for describing the position of an entity in space. This can be easily generalised to any number of dimensions.
Constants	
$adaptNotif$	The time it takes for an entity to be notified of a message delivery area. This is assumed to be strictly positive.
$msgLatency$	The time between initiating the sending of a message and its actual delivery. This is assumed to be strictly positive.
s_{max}	We assume a maximum speed at which entities may travel.
Functions	
$trans(m)$	The worst case time it takes for an entity in a mode m to reach a fail-safe mode.
$period(m)$	The time between successive broadcasts from an entity in mode m. This is assumed to be strictly positive.
$fs(m)$	Predicate on modes determining which modes are fail-safe. The predicate is true if the mode m is fail-safe.
$m \rightsquigarrow m'$	The mode transition relation, which relates pairs of modes when one can transition to another. Whenever the modes m and m' are related, m' is often referred to as a successor mode of m.
$tt(m,m')$	The time it takes to transition from one mode to another. This function is only defined on modes that are related via the mode transition relation.
$fSucc(m)$	We assume that every mode has at least one fail-safe successor. This function maps every mode to a successor mode that is fail-safe.
$minDistComp(m,m')$	The *minimum distance of compatibility* function between two modes, which is explained later in Definition 3.3.
Conditions	
$\forall\, m,m', minDistComp(m,m') = minDistComp(m',m)$	The *minimum distance of compatibility* function is symmetric.
$\forall\, m,m', fs(m) \implies minDistComp(m,m') = 0$	Fail-safe modes have a minimum distance of compatibility of 0 with any other mode. Note that due to symmetry the order of the arguments do not matter here.

state when values are given to all of its state variables. A collection of entities and their corresponding states constitutes a system state or simply a state.

From here, it is noted that the behavioural model should aim to characterise the evolution of system states. The model chosen here is borrowed from the theory of hybrid systems, which are discussed in Section 2.3.4. In this paradigm, systems progress according to two different types of transition. The first type of transition models the continuous evolution of state variables with time e.g. the curved trajectories of the entities' positions through space. The second type of transition models a discrete software event e.g. the execution of a single statement by the program running on board some entity. It is assumed to be atomic and instantaneous i.e. it does not take any time to occur. Of course, in reality, even a software action will take time to complete, but it is generally accepted to consider software actions as instantaneous because the time taken for them to occur is usually considered negligible relative to other events e.g. the movement of physical objects.

It is assumed that the software components can change the modes of the corresponding entities and initiate the sending of a message. Messages are then broadcast to a certain radius in the environment of an entity. Any entity within range of a broadcast receives the message and its software component can process the contents of the message. Feedback of coverage is provided to the sender according to the Space-Elastic Model, as discussed in Section 3.3.1.

3.3.3 Safety

According to the original model, the essential strategy by which entities ensure safety is: to "ensure that the modes of elements are always compatible when any of them is in the safety zone of the other" [Bouroche, 2007]. The interpretation of this statement here leads to the assumption of a minimum distance of compatibility between any pair of modes. This is characterised by a function which is defined in Definition 3.3. A slight variation on this function is given in Definition 3.4.

Definition 3.3 (Minimum Distance of Compatibility)**.** The function $minDistComp(m, m')$ gives the minimum distance by which two entities $\mathbf{E}$ and $\mathbf{E}'$, in modes m and m' respectively, must be separated in order to guarantee that they do not cause a violation of safety. At and above a distance of $minDistComp(m, m')$ apart, the entities $\mathbf{E}$ and $\mathbf{E}'$ are guaranteed to be compatible. The function $minDistComp$ is referred to as the minimum distance of compatibility. It is system specific and is thus left uninterpreted in this general model. ■

Definition 3.4 (Max-min Distance of Compatibility)**.** The max-min distance of compatibility for a mode m is the maximum of all the minimum distances of compatibility over all other modes m'. It is defined as follows:

$$d_{max}(m) \stackrel{\text{def}}{=} setMax(\{minDistComp(m, m') | m' \in \mathbb{M}\})$$

Here $setMax$ is simply a function returning the maximum value of a finite set; the set of modes is assumed to be finite. ■

Notice the generality of these functions. The nature of the modes involved is irrelevant. All that matters at this level of detail is that every pair of modes is given a distance beyond which they are guaranteed to be compatible. Building upon this definition, a general safety condition can be formulated. Safety is interpreted for a given state i.e. at a certain point in time. Bearing this in mind, a definition of safety is given in Definition 3.5.

Definition 3.5 (Safe State)**.** A state is said to be safe whenever every pair of entities in that state are separated by a distance greater than or equal to the minimum distance of compatibility for their respective modes. ■

It is conceivable that safe states can transition to unsafe states. For example, imagine a scenario in which a hedge-trimming vehicle is approaching the end of a hedge that it is trimming and there is a traffic light located just beyond the hedge. The traffic light and vehicle might be considered entities in the system with a certain minimum distance of compatibility for when the hedge trimmer is on. The violation of safety would be that the protruding hedge trimmer would collide with the traffic light causing damage potentially to both. The hedge

trimmer being off could then be considered a fail-safe mode. Without the correct constraints on behaviour, the hedge trimmer could simply drive directly past the traffic light with the hedge trimmer still on and protruding, and there would be a collision. Evidently this would constitute a violation of safety. On the other hand, before the collision the system is safe, showing that it is possible for a safe state to transition to an unsafe state.

The above example is informal and somewhat contrived, but it demonstrates the need for entities in a system to obey certain behavioural constraints in order to maintain safety. The basic strategy from the original model by which safety is maintained is that entities ensure compatibility within certain zones of each other. This key idea is maintained in the revised model, though the specifics are slightly different. For example the notion of incompatibility is simplified in the revised model. Also, compatibility in the revised model is ensured via a single contract, in contrast to the original model which has three contract types. The details of this revised contract follow in Section 3.3.4.

3.3.4 A Modified Contract

Recall that there are three types of contract in the original Comhordú model. To simplify matters, just one type of contract is considered in the revised model: a contract with transfer without feedback. Furthermore, in light of the modifications made to the concepts of safety and scenario, and the introduction of a template for a behavioural model, the specifics of this type of contract need to be revised. Recall that a contract is a set of rules governing the behaviour of entities. An informal presentation of the rules for a revised contract with transfer, without feedback is given below. It is noted that these rules are a modification of the original contract without transfer as presented in [Bouroche, 2007].

init To begin with, it is assumed that all entities are in fail-safe modes. That is, the system is initially assumed to be trivially safe.

transfer$_1$ Every time an entity in a fail-safe mode desires to enter a mode that is non-fail-safe, it must first warn other entities a certain predetermined time in advance of entering this new mode. The warning must be carried out by periodically broadcasting the current position of the entity along with the desired mode of operation. The duration of time over which this periodic warning must occur is parametrised on the desired mode in question. The period of the broadcast is also parametrised on this mode.

transfer$_2$ Once the warning time has elapsed, an entity may enter the desired mode of operation, which then becomes its current mode.

transfer$_3$ As long as an entity remains in a non-fail-safe mode, it must continue to periodically broadcast to surrounding entities in the same fashion as during the warning phase. In this case, the entity is said to be acting in the mode in question.

transfer$_4$ An entity that intends to switch from one non-fail-safe mode to another must first warn other entities in much the same manner as is done according to **transfer**$_1$. The only difference is that simultaneous broadcast of both modes is necessary during the warning phase. The mode into which the entity desires to switch is referred to as the next mode while the current mode of operation is simply referred to as the current mode. The current and next modes must be periodically broadcast simultaneously until the warning time of the next mode elapses. At that point, a mode switch can occur, whereby the current mode is replaced by the next mode.

adapt$_1$ If an entity receives a message from another entity that threatens a potential incompatibility with the current mode of operation, then it must adapt accordingly by immediately initiating a transition to a fail-safe mode. The notion of "potential incompatibility" is not defined here, but depends on the distance between the current position of the entity and the position embedded in the message, which is the position of the sender at the time of sending. A more complete definition can be found in Definition 5.5.

adapt$_2$ Similarly, if an entity acting in some mode receives a notification that one of its messages has been delivered to an insufficient coverage, then if must immediately adapt by initiating a fail-safe transition. The exact notion of "insufficient coverage" is not elaborated here; for a definition of its negation, i.e. sufficient coverage, the reader is referred to Definition 5.4. However it is mentioned here that this notion depends on the *minimum distance of compatibility* function and the mode being broadcast.

abort$_1$ If an entity in the process of switching modes receives a message from another entity that threatens a potential incompatibility, relative to the next mode, then it must abort the transition to the next mode.

abort$_2$ If an entity in the process of switching modes receives a notification of coverage degradation relative to the next mode, then it must abort the transition to the next mode.

Note in the above rules it is assumed that threats to the current mode are given priority over threats to the next mode. That is, say a coverage degradation or potential incompatibility occurs relative to both the current and next modes. Then the entity conservatively takes the most extreme action by initiating a transition to a fail-safe mode. That is, the threat to the current mode is considered more severe and addressed appropriately.

The first rule specifies the initial condition that must hold. Without this, an arbitrary unsafe state could be chosen as the initial state under consideration, which wouldn't make sense. The next four rules characterise the transfer primitive. That is, an entity transfers responsibility to its neighbours to react to its behaviour by first warning the neighbours, switching to the potentially hazardous mode, and then continuing to notify its neighbours of continued operation in that mode. Switching to a new mode necessitates further warning to other entities before the mode switch can occur. The next two rules constitute the adaptation primitive. That is, upon detecting either a coverage degradation or a possibly incompatible neighbouring entity, an entity must react by conservatively switching its mode to a fail-safe mode. The final two rules constitute the delay primitive from the original model, which in this case is realised by an entity aborting its current mode transition. An entity wishing to switch its mode must cancel doing so if either a potentially incompatible entity enters its proximity or a coverage degradation occurs. Note that the incompatibility and degradation in this case are relative to the next mode rather than the current mode.

These rules closely resemble those of a contract with transfer without feedback given in [Bouroche, 2007], the main difference being that there is no distinction made between "responsible" and "other" entities. That is, *all* entities are responsible for following the same contract, which incorporates within it all three coordination primitives of responsibility transfer, adaptation and delay. In comparison to the original model though, the adaptation primitive is somewhat over-conservative. That is, in the original model adaptation does not necessarily entail a transition to a fail-safe mode but could instead imply a transition to a non-fail-safe mode that is no longer potentially incompatible. However to include this behaviour would significantly complicate the model.

As a consequence of the pursuit of greater precision towards the development of a formal model, this specification of a contract comprises more rules than the original specification. To elaborate, there are eight rules constituting this specification in comparison to the original contract specification, which contains only four rules. This highlights the natural increase in detail that usually accompanies an increase in precision. The detail further increases when a full formal specification is given in the following chapters.

3.4 Assessment of Revised Model

It is now time to assess the revisions made to the Comhordú model. The assessment begins with a general discussion of the model in Section 3.4.1. Section 3.4.2 details the components from the original model that are omitted in the revised model. In Section 3.4.3, an account is given of the elements of the original model that are modified in the revised model. Where relevant, various consequences of these omissions/modifications are examined, and in some cases justification is given for the choices made to omit/modify the details in question.

3.4.1 General Discussion of Revised Model

On the whole, the choice to adopt a unified model encompassing all scenarios entails the compromise of a limited expressiveness in the behaviour of systems. That is, only the "lowest common denominator" behaviours, so to speak, can be included in a general model. The alternative, to provide separate semantics for every scenario, seems unsatisfactory as it detracts from the notion of a unified coordination model. Perhaps though a hybrid between the two approaches is possible i.e. certain "classes" of scenario could be identified and different behavioural models provided. For example, a class of scenario could model the general case of a road intersection, but the number of roads/entities, could be left unconstrained. A behavioural model in this case could exploit the assumption that vehicles travel within lanes. Nonetheless, the direction chosen for this work is to provide a unified model; application specific approaches such as the junction example are left to future work.

As is stands, there are certainly opportunities to extend the revised model to include more features from the original model. However, as a first step towards a formally specified model it nonetheless preserves a reasonable level of key concepts from the original model. In defence of the simplifications made, it may be observed that with the addition of rigour, that is a necessary step towards formalisation, the level of detail of a model tends to increase significantly. For example, while behaviour is implicit in the original model, even to include a rough framework of behaviour in the revised model is non-trivial as can be seen from the reasoning outlined in Section 3.2.1 and the behavioural framework sketched out in Section 3.3.2. In short, for reasons of practicality, some details from the original model are excluded from the revised model.

It is also worth noting here that the revised model is still informal. Particularly the behavioural model is not given any precise definition. While this still leaves certain aspects vulnerable to ambiguity, the unified and simplified nature of this revised model renders it amenable to formalisation. In fact, its very conception is solely towards the ultimate goal of developing a fully formal model. In any case, any ambiguities in this revised model do not matter all that much, because the following chapters set down a rigorous formal model embodying the entire revised informal model. That is, all concepts are ultimately defined formally and unambiguously.

3.4.2 Omissions

A number of elements from the original model are wholly omitted from the revised model. These are outlined below.

- There is no contract without transfer and no contract with transfer with feedback in the revised model. The only contract modelled is the contract with transfer without feedback.
- The sensor and indirect communication model is not included in the revised model. The reasoning behind this omission is that to include a sensor and indirect model on top of all the concepts already modelled would introduce too much complexity into the model.
- Priorities and priority lists are omitted. It does not seem that this omission is that detrimental to the integrity of the model. Priority lists simply specify which modes are more important than others. In this model, where there is no such ordering imposed upon modes, behaviour is more general and so subsumes all the behaviours that would be present in a model with priorities. From the point of view of proving a safety property, as is the current goal, any over-inclusion of behaviours such as this is not problematic.
- Passive elements are not modelled. That is, elements such as, say pillars, roadsigns etc. that would be present in most real-world systems are not present in the states of the revised model. Only the programmable elements, referred to as the entities, are included. The choice to omit passive elements follows from the fact that the sensor and indirect communication model is omitted. That is, in the revised model entities do not have the capability to sense objects in their environment, and so safety relative to these objects could never be achieved in this model. There is a potential loss of integrity resulting from this

omission. Most real-world scenarios do contain passive elements that entities need to be able to negotiate. However, the inclusion of a sensor model is left as a separate concern. Perhaps the sensing of a potentially hazardous obstacle could elicit a reaction to a fail-safe mode, in much the same manner as an entity reacts to degraded coverage or possible incompatibility with another entity.

- The original Space-Elastic Model includes a parameter called *present*, which hasn't been discussed until now. This parameter is not included in the revised model. It denotes the amount of time it takes an entity to "join" the coverage of another entity, so to speak. That is, if an entity $\mathbf{E_1}$ enters the coverage of another $\mathbf{E_2}$, it will not be able to receive messages from the entity $\mathbf{E_2}$ until it has remained within the coverage of $\mathbf{E_2}$ for at least *present* time units. Once $\mathbf{E_1}$ has been in the coverage of $\mathbf{E_2}$ for at least this amount of time, then it is guaranteed to receive all messages from $\mathbf{E_2}$ for as long as it remains in the coverage of $\mathbf{E_2}$. The entity $\mathbf{E_1}$ may leave the coverage of $\mathbf{E_2}$, because of either relative movement between the entities or because of a coverage degradation of $\mathbf{E_2}$, or both. As soon as this happens, it must re-join the coverage in order to receive further messages, which again takes a time of *present*.

  In contrast, the revised Space-Elastic Model provides to an entity the area to which each of its messages was guaranteed to be delivered. It is now demonstrated why this simplification can be made. Imagine a sender entity sends a message, which is delivered at a certain time t. Now, consider the coverage of the entity at all points in time from the time $t - present$ up until t. The coverage may fluctuate over this interval. Still, there is guaranteed to be a minimum coverage, call it x, reached within this window of time. The coverage x may be null, i.e. a circle with a radius of 0. Recall that coverage is being modelled by a circle characterised by the position of the sender and a radius.

  Now, subtracting $2 * s_{max} * present$ from this minimum coverage, yields a value, call it r. That is $r = x - 2 * s_{max} * present$. Depending on the value of the minimum coverage the value of r may or may not be positive. If it is positive, then it characterises the area about an entity within which all entities are guaranteed to receive the message at the time of delivery. To see why this is the case, imagine an entity within r, if this value is indeed positive. Then over the last *present* time units, this entity must always be within a distance $r + 2 * s_{max} * present$ of the sending entity; here the term $2 * s_{max}$ denotes the maximum relative speed between any two entities. Now, this is exactly the minimum bound on coverage x previously introduced. Hence any such entity is guaranteed to always have been in the coverage of the sender for at least *present* time units, and thus is guaranteed to have received the message.

  By keeping track of its past coverage and performing the necessary calculations according to the above reasoning, an entity is capable of converting feedback on its coverage, as defined in the original model, to information about the area to which a message is guaranteed to have been delivered. Rather than explicitly include the original model of coverage and all of this reasoning, coverage in the revised model is considerably simplified to mean just the area to which a message is delivered. That is, the above calculations are assumed to take place on-board an entity behind the scenes and are not explicitly modelled. This yields a more concise model.

  This reasoning is captured in Figure 3.1. Shown in the diagram is the original model of coverage, referred to as the *actual coverage*. Also shown is the entity's knowledge of its coverage, which lags the actual coverage by a value of $adaptNotif$. A message is assumed to be delivered at time *delivery time*. All entities within the *simplified coverage* receive this message. This is because, based on the preceding reasoning, these entities will all have been within the coverage of the sender for at least *present* time units. To elaborate, the positions of these entities are bound by the maximum relative speed between entities, i.e. they will always be under the *entity position bound* line. Since this line is defined to be never above the *minimum coverage*, which is the lowest point the coverage reaches, the entities must be in the coverage for the entire window of size *present*. Moreover, at the *notification time*, an entity will know of the radius to which it its message was delivered because by this time it will have knowledge of the coverage over the entire window of size *present* preceding the *delivery time*. From this, it can easily

calculate the minimum coverage over this window, and in turn the simplified coverage. Thus, it is safe to assume the simplified model. Note that in the diagram a non-null simplified coverage is assumed.

- Entity types do not feature in the revised model. Given the generality imposed by the demands of unification, entity types seem like a superfluous detail for a revised model. That is, behaviour is unified for all entities and so it seems redundant to impose subtypes upon the set of entities. An alternative approach, i.e. to model different classes of scenario, could conceivable utilise entity types to define different behaviours. It is worth noting that the explicit omission of entity types in the revised model does not preclude the existence of entity types, rather it just abstracts from the details of such.

- Mode invariants are excluded from the revised model. These are constraints on the evolution of state variables based on the mode of an entity. For example, the upper bound on the speed of an entity might depend on its mode. The omission of these mode invariants, like many other omissions, simply allows for more behaviours in the model. That is, all the behaviours possible with mode invariants are included in a model without invariants, because the latter is unconstrained. Furthermore, even if mode invariants were to be included, they would be limited in that the only state variable other than mode that features in the revised model is position. Speed does not even feature as an explicitly modelled state variable. Thus mode invariants would be limited to constraints on the position of an entity based on its mode. In any case, from the perspective of safety, in its revised form, these invariants are superfluous.

- Goals are not included in the revised model. These could potentially be achieved by a higher level set of behaviours that could be somehow composed with the contract. The composition of Comhordú with a higher level protocol is outlined as an avenue for future work in Section 8.5. Like many other details from the original model, goals are irrelevant from the point of view of safety, in its revised form; thus they are neglected in the revised model.

- Solvability is not covered in the revised model. Recall from the original model that a scenario is solvable if there exists a set of behaviours for its constituent entities such that these behaviours guarantee safety for that scenario. Depending on the allowable behaviours of entities within a particular scenario, and the nature of the safety constraint, some scenarios may not be solvable. For this reason, only solvable scenarios in the original model are considered for the application of contracts. However, in the revised model, due to the unification approach therein, the very notion of a scenario is not explicit. Furthermore, the safety constraint is homogeneous across all states in the state-space, parametrised only on the *minimum distance of compatibility* function. For these reasons, it does not make sense to talk about solvability in the revised model.

3.4.3 Modifications

Most features from the original Comhordú model that are included in the revised model are modified in some sense. These features are listed below, and the modifications made are given some examination.

- The contract with transfer without feedback is altered in light of the modifications made to the safety constraint and the new behavioural model introduced. The new specification of this contract is also more detailed in light of the increased rigour of the revised model. The presentation of this modified contract along with its comparison to the contract in the original model can be found in Section 3.3.4.

- The safety constraint is simplified to a general construct. To elaborate, there is only one safety constraint parametrised on the *minimum distance of compatibility* function rather than a language for expressing various different safety constraints. In particular, this function allows only for the specification of binary incompatibilities, in comparison to the original safety constraint language which allows for the expression of higher arity properties i.e. properties involving more than two entities. Still, even with the current choice of safety constraint, flexibility is afforded to a system designer in the specification of the *minimum*

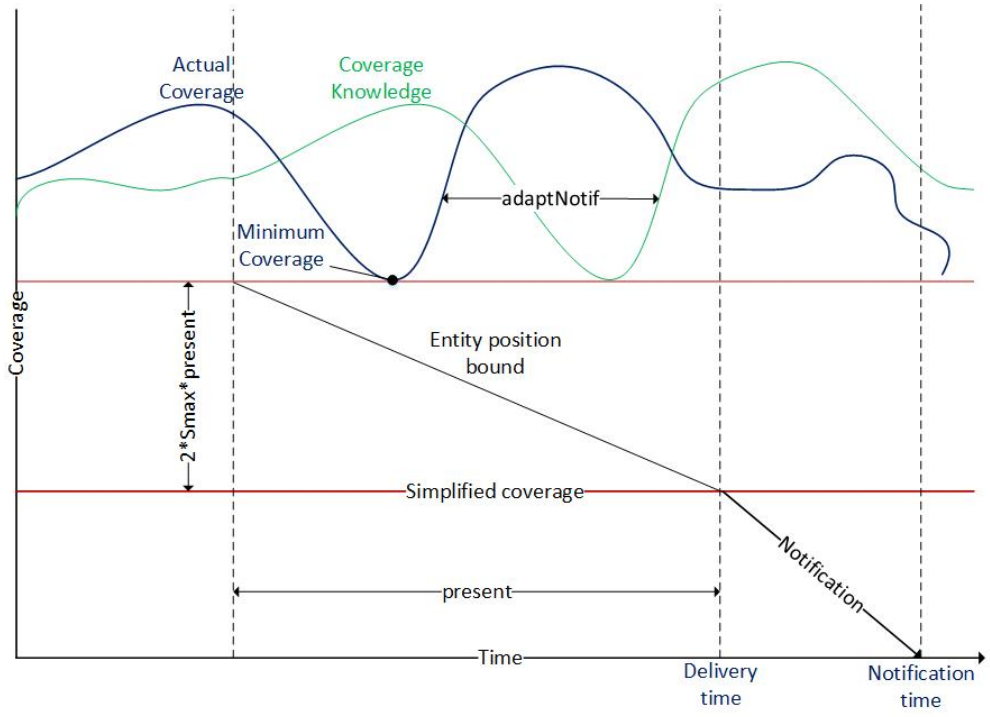


Figure 3.1: Depiction of the new model of coverage as a simplification of the original model.

distance of compatibility function. The choice to simplify the safety constraint in this manner stems from a need for unification, as outlined in Section 3.2.2.

- The abstract notion of a state variable is abandoned in favour of assigning each entity with just two state variables: position and mode. This does not mean that entities cannot have state variables other than position; rather these other variables are of no relevance with respect to safety as it is defined here and so they do not feature explicitly in the model.

- The behavioural model presented in the revised model is explicit in comparison to the original implicit inclusion of behaviour. Nonetheless, it is still informal and somewhat incomplete. To elaborate, the behavioural model simply introduces the notion of a state-space, a transition relation over this state-space and the idea from hybrid systems that transitions can be either discrete or continuous. It does not give any characterisation of this transition relation. However, to do so here is deemed unnecessary as the formal model of the ensuing chapters does provide an explicit behavioural characterisation.

- The Space-Elastic Model is simplified somewhat. To begin with, coverage is abstracted to just a circle rather than an arbitrary area about an entity. However, this appears to be an acceptable approach even in the original model: “Proximities can be of any shape and can be defined either absolutely (via gps coordinates), or relatively around the entity (using an anchor point and a size)” [Bouroche, 2007]. There is also the replacement of continuous coverage updates with just message delivery updates and the omission of the parameter *present*, both of which are justified in Section 3.4.2. Furthermore, coverage notification is assumed to occur exactly after $adaptNotif$ time units rather than at any point within this time frame. This restriction does not compromise the original model. It can be interpreted to mean that an entity simply buffers any “early” delivery notifications that occur before $adaptNotif$ and consumes them once $adaptNotif$ time units have passed.

- There is no longer any explicit notion of a scenario. Instead there is just one state-space subsuming all the possible states of all possible scenarios, parametrised on the *minimum distance of compatibility* function. Hence the choice of scenario is replaced by the choice of a concrete *minimum distance of compatibility*

function and a start state in the state-space parametrised on this function. The behaviours of this state are characterised by the transition relation that is assumed to exist over the whole state-space.

- All entities in the revised model are responsible all the time and follow the same contract. In contrast, the original model only stipulates that some entities need to be responsible and different contracts are followed in general. Since responsible entities must be capable of bounded transitions to fail-safe modes, and all entities in the revised model are responsible, it is necessary that fail-safe modes are reachable within a bounded time by every entity in a non-fail-safe mode.

- The constant s_{max}, denoting the maximum absolute speed for an entity, replaces the more detailed Vmax(m) given in [Bouroche, 2007], which is the maximum speed relative to an entity in mode m. This parametrised version of maximum speed would possibly allow for slightly tighter bounds to be deduced. For example, on receipt of a message from another entity, the receiver might be able to deduce an upper bound on the speed of the sender based on the mode contained in the message. However, since the mode may have changed since the message was sent, it is not at all obvious that this bound could even be deduced. Hence the inclusion of a parametrised maximum speed seems superfluous and is abandoned in favour of the simpler notion of a single upper bound on speed for all entities.

- Mode transitions for different entity types cannot explicitly exist because the notion of entity type is omitted. However, this does not preclude their implicit existence. As previously mentioned, entity types could be superimposed on the "flat" set of entities. Separate mode transition relations could then exist as disjoint sub-relations of the unified mode transition relation. Since the existence of different entity types or mode transition relations has no bearing on safety, this does not seem like an issue of major importance.

3.4.4 Assessment of Revisions: Summary

To summarise it may be noted that many omissions and modifications in the revised model exclude detail that is irrelevant to the safety constraint. This is justified by the fact that only the safety constraint property is of interest here. It is conceivable that extra detail might be necessary for the exploration of other properties. For example, the inclusion of goals and priorities in the model would undoubtedly be necessary if conditions on the fulfillment of the most important goals were of interest. The lack of any explicit inclusion of such features, however, does not preclude the applicability of the model to systems that *do* exhibit these features. They operative word here is "explicit". Just as a theorem about all natural numbers is true for all odd numbers despite any explicit reference to odd numbers in its statement, so too can many systems with additional features inherit the properties proved in this model, despite the explicit occurrence of such features in the model.

3.5 An Abstract Formalisation of Comhordú in UPPAAL

This section presents an initial endeavour towards formalising Comhordú using the UPPAAL model checker. The presentation given here is intentionally brief; for the full paper on this topic the reader is referred to [Bhandal et al., 2011a]. Much of the remainder of this section is taken from [Bhandal et al., 2011a], particularly Section 3.5.1 which describes the UPPAAL model itself. Following this is a discussion in Section 3.5.2 evaluating the UPPAAL model, in particular its shortcomings, establishing a need for further work i.e. the remaining content of this thesis.

3.5.1 Summary of Comhordú in UPPAAL

UPPAAL [Larsen et al., 1997] is a model checking tool which allows Timed Automata to be built, simulated, and analysed[1]. Timed Automata TA constitute a formalism for describing real-time systems. A timed automaton has the following basic components:

- Clocks: A finite number of real-number valued variables.
- Locations: These can be thought of as the discrete states of the system, whereas a full state is qualified by a location and a "clock valuation" i.e. a mapping from all the clock variables to concrete numerical values.
- Edges: These model transitions between locations. Each edge is assigned up to 3 labels. The first is a delay guard, a condition upon the clocks which must be true in order for the transition to be taken. The second label is an action representing something the automaton "does" during the transition. The third label indicates a number of clocks to be reset following the transition.
- Invariants: Conditions on locations involving the clocks that cannot be violated during a "run" of the system. Placing an invariant at a location can be used to force an automaton out of that location before the invariant becomes falsified.

For a more detailed introduction to Timed Automata, the reader is referred to [Alur and Dill, 1994].

The abstract model of Comhordú consists of UPPAAL automaton templates. These are automata specified in terms of symbolic parameters. Instantiating such a template, i.e. providing concrete values for all its parameters, gives rise to an automaton. Instantiation is only necessary at the model checking phase rather than the modelling phase, which allows the same template to be reused any number of times, potentially with different parameters each time.

The main template in this UPPAAL model is shown in Figure 3.2. This models an entity in the system. The integer parameters to this template are $treact$, $twait$, and $period$. The parameter $treact$ is the maximum time it takes an entity to reach fail-safe mode, while $twait$ is the maximum time an entity must wait before entering a non-fail-safe mode. The length of time between successive broadcasts from an entity is denoted by $period$.

Recall from Section 3.3 that the behaviour of the Comhordú protocol is to broadcast messages periodically containing the current/next mode of operation. This behaviour is captured by self-looping edges in the locations "acting"/"preparing" respectively. The former represents the entity in a stable mode while the latter represents the entity as it transitions into some mode.

The locations "failsafe" and "good" both represent the entity in a fail-safe mode, except that for the former coverage is bad while for the latter it is good. A similar distinction is made between the locations "reactingBad" and "reactingGood", both of which represent the entity's state as it transitions from an acting mode to a fail-safe mode.

Coverage degradation/improvement are both modelled by the action "adaptNotif" which can be seen to label a number of edges. The action can be thought of as representing a toggle between good and bad coverage. This action comes from an external "environment" component rather than from the entity itself.

Transfer messages are modelled by the channel *msgIn*. Along this channel, entities can both send and receive messages. Sending a message on this channel warns the other entity that critical zone activity taking place or that it is imminent. An entity in a critical zone immediately reacts to such a message by initiating a transition to a fail-safe mode. An entity already in a fail-safe mode ignores such messages.

In the original paper [Bhandal et al., 2011a] a number of properties are verified for this UPPAAL model. The verification is done for an instantiated system containing two entities and a number of other components modelling the environment. The properties and a brief account of their meanings are outlined in the following.

[1] There are numerous introductory articles and tutorials e.g. [Behrmann et al., 2004] available on UPPAAL. Also, at the time of writing, the tool may be downloaded for academic use from the official UPPAAL webpage: http://www.uppaal.com.

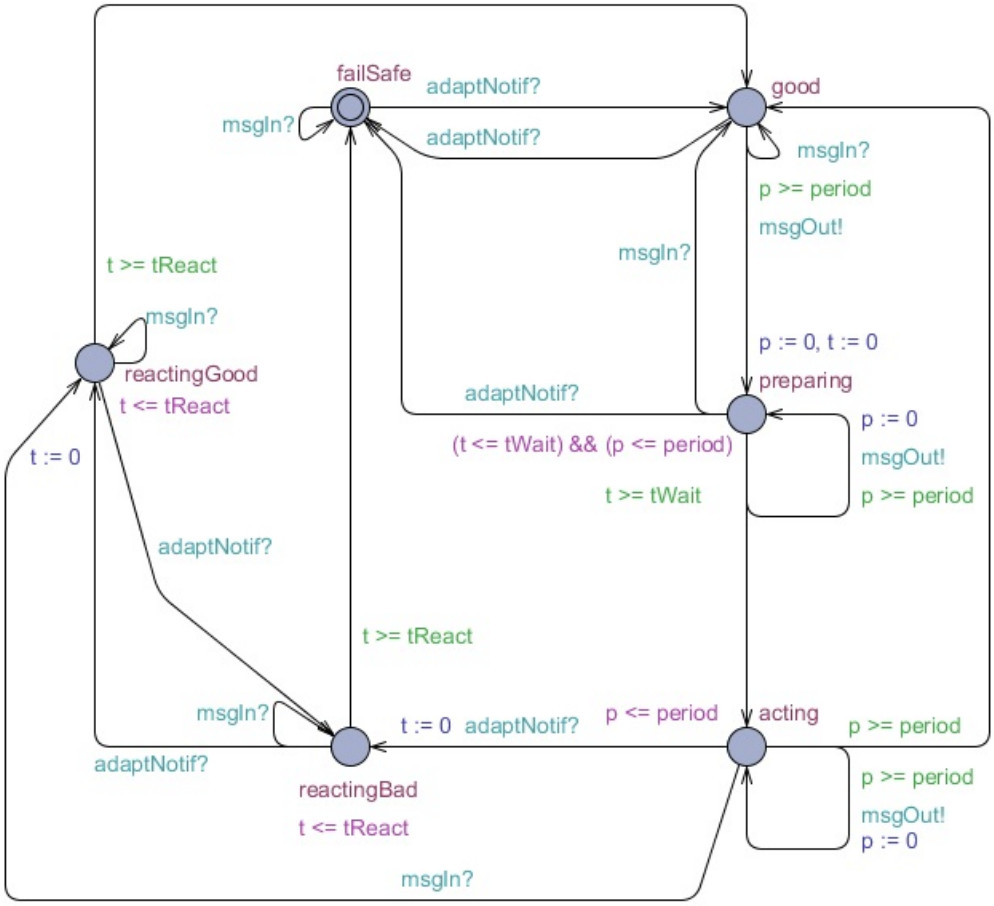


Figure 3.2: The Entity Template.

- *A[] not deadlock*: A desirable property of most reactive systems such as this one is that it is without deadlock. Deadlock is a property satisfied by any system state which has no outgoing transitions. Transitions here refer to either discrete transitions, modelled by edges, or timed transitions. In the latter case, the location remains the same but time passes on, changing the overall state. The entire property stated here uses universal quantification over all states of the system with *A[]*. Hence the property in full says that no state of the system is ever deadlocked. Universal quantification in this manner is used in the remaining properties.

- *A[] not (entity1.acting && entity2.acting)*: This is the first of six mutual exclusivity properties. These properties are highly important in that their satisfaction guarantees the safety constraint in an abstract sense. That is, both entities should not be in the non-fail-safe mode at the same time. The verification of this property guarantees that in all states of the system, both entities are not acting at the same time. The following five properties constitute the other mutual exclusivity properties.

- *A[] not (entity1.acting && entity2.reactingGood)*.

- *A[] not (entity1.acting && entity2.reactingBad)*.

- *A[] not (entity1.reactingBad && entity2.reactingBad)*.

- *A[] not (entity1.reactingBad && entity2.reactingGood)*.

- *A[] not (entity1.reactingGood && entity2.reactingGood)*.

- *E<> entity1.acting*: This property asserts that is is *possible* for an entity to reach an acting state, i.e. it is a progress property.

Note that the need for six mutual exclusion properties here rather than just one stems from the fact that an entity is in a non-fail-safe mode if it is either acting, reactingBad or reactingGood. Since there are two entities this constitutes $9 = 3 \times 3$ combinations, but 3 of these can be discarded due to symmetry between entity1 and entity2.

Remark 3.6 (New Properties)**.** The progress property and the final three mutual exclusivity properties do not appear in the original paper. That is, the properties *A[] not (entity1.reactingBad && entity2.reactingBad)*, *A[] not (entity1.reactingBad && entity2.reactingGood)*, *A[] not (entity1.reactingGood && entity2.reactingGood)* and *E<> entity1.acting* are not included in the paper. However, they have since been verified with the UPPAAL engine and so are included here. ▲

This model abstracts or simplifies a lot of the detail of the revised Comhordú model as follows.

- Positions are completely omitted. Consequently then movement is also omitted, as any model of movement would include within it a model of position.
- There are only two modes in the system: fail-safe and non-fail-safe. The non-fail-safe mode is characterised by the locations acting, reactingBad and reactingGood. The fail-safe mode is characterised by the locations failSafe, good and preparing.
- Coverage is abstracted as being either good or bad. The justification for this choice of abstraction is that since there is only one non-fail-safe mode, the only relevant property of coverage is that it is sufficient for that mode or not. If more modes were to be included in the model then a more detailed model of coverage would be needed to account for the different notions of sufficient coverage in relation to the different modes.
- Since there is only one non-fail-safe mode of operation, there cannot be any model of the behaviour of switching from one non-fail-safe mode to another. Rather, an entity remains in *the* non-fail-safe mode until either it decides to transition back to a fail-safe mode of its own accordance, or it is forced to adapt by initiating such a transition because of a coverage degradation or possible incompatibility.
- There is no content to the messages that are sent between entities. This is an inevitable consequence of previous omissions of detail. That is, while a message should contain the position of the sender and its mode, neither of these data feature in messages of this model because position is not modelled and there is only one non-fail-safe mode, so to include it would be superfluous.

3.5.2 Discussion of Comhordú in UPPAAL

The abstract formalisation of Comhordú in UPPAAL constitutes significant progress in the overall formalisation effort of Comhordú. In particular, it solidifies the intuition that periodic message broadcast can enforce mutual exclusion. However, there are a number of limitations to the model.

Firstly, the system that is verified is concrete rather than abstract. There are only two entities involved and the parameters are given precise integer values. What comes to mind here is that perhaps the system could be verified for a number of other instantiations of the abstract templates i.e. for different parameters and a different number of entities. Certainly, this would increase confidence in the UPPAAL model, but this option has been postponed in favour of the pursuit of a more thorough system description.

In any case, instantiations aside, this model even in its abstract form is limited in it expressivity. This is clear from the abstractions outlined at the end of the previous section. In short, the loss of detail incurred by making these abstractions is unacceptable as a conclusive result of the pervading question of this thesis, namely whether Comhordú is correct in terms of the guarantee of safety implied by contract adherence.

A key factor lending to the insufficiency of this result is that there is no notion of position or mobility at all in the model. Considering that these are essential characteristics of Comhordú, it is unacceptable to omit this detail in any faithful model. A question that comes to mind here is, does there exist a “hack” to encode position

in UPPAAL? On inspecting the UPPAAL modelling language, the answer to this question seems to be "no". An initial thought that comes to mind is to model space as a finite set of discrete points, and represent these with locations, but this idea must be instantly discarded as any such attempt would lead to an enormously large state-space even for a small amount of discrete points in the space.

Still, although this is not an entirely acceptable result within the scope of the current thesis, it is somewhat promising. The machine-verified proof shows that periodic broadcast implies mutual exclusivity from a critical zone of operation. This consolidates original intuitions that this is the case and increases confidence that the result holds for the more complex model including an arbitrary number of modes and a model of mobility for entities. Looking at this in a slightly more formal manner, the UPPAAL model can be viewed as a very special instance of the full model with only two entities, two modes, and fixed timing parameters; also the entities are always assumed to be within a possibly incompatible distance of each other. Proving this special case may not be wholly satisfactory, but it is often promising when a special case of a problem is shown to be true towards gaining faith in the truth of the general case; this point is argued elsewhere: "Specialization is often useful in the solution of problems" [Polya, 2004].

In summary, the UPPAAL model is fully verified in terms of a mutual exclusion property, but it is too simple. This simplicity is the result of the limitations of UPPAAL, and indeed model checking in general, chiefly the finiteness and concreteness that must be imposed on a system in order that it be eligible for verification. These limitations have steered formalisation attempts for Comhordú away from the model checking domain and towards the more expressive domain of process algebras. An initial encoding of Comhordú in a process algebra is given in [Bhandal et al., 2011b]. This is superseded by the current process algebraic model given in Chapter 5.

3.6 Methodology

This section briefly introduces the chosen methodology that forms the content of the remaining chapters. The purpose of this methodology is to formalise the revised Comhordú model given in this chapter and verify that the specification of a contract with transfer is correct with respect to safety. The methodology departs from the approach of model checking. The decision to make this departure stems from the limitations encountered with the UPPAAL model checker, as described in Section 3.5. An outline of the current choice of methodology is as follows.

- A formal specification language is presented in Chapter 4. This language constitutes a precise behavioural model for real-time systems of mobile entities with wireless communication capabilities. The language draws inspiration from existing languages but is novel to this work. The choice to devise a new language is a result of the limitations of expressiveness imposed by existing formal specification languages. The language is split over a number of layers or tiers which are formulated as sub-languages.

- A protocol is presented in Chapter 5 emulating the contract with transfer specified in Section 3.3.4. The design of the protocol is first sketched informally and then given a formal specification in the software fragment of the overall formal specification language previously mentioned. An argument is put forth in Section 3.6.1 in favour of adopting this protocol as an alternative to the contract with transfer.

- A proof is given in Chapter 6 that the protocol is correct i.e. that it guarantees safety. That is, given certain initial conditions on a state of entities, such that all entities are running the protocol, all future states are guaranteed to be safe.

- An encoding of the language, protocol and proof into the Coq proof assistant is undertaken and described in Chapter 7. This adds rigour to the aforementioned specification and proof.

3.6.1 The Comhordú Protocol: an Alternative to the Contract with Transfer

Recall from Section 3.1.2 that a fundamental element of the original Comhordú model is the notion of responsibility. In short, responsibility entails that for every possible incompatibility in a scenario, a certain entity is elected to ensure independently that the incompatibility never becomes true. The chosen entity is called the responsible entity. This idea of responsibility is elaborated by the concept of a contract. A contract is a set of conditions on the behaviour of an entity that it should obey i.e. entities that are assigned contracts become responsible for obeying their contracts. In a given scenario, contracts are assigned to entities with the intention that adherence to the contracts by all responsible entities implies safety in that scenario.

Now, recall that in the revised model, safety is specified in terms of an uninterpreted function called the minimum distance of compatibility. Also, the idea of a scenario is subsumed by a unifying model of behaviour which incorporates a single state-space of all possible states of all possible scenarios and a transition relation over this state-space. In light of these modifications to the notions of safety and scenario, what then is the meaning of responsibility in this new setting?

Well, an initial idea for interpreting responsibility taking into account these revisions would be to use the existing notion of a contract. However, on inspection of the various types of contract, it becomes apparent that the behavioural stipulations of contracts with transfer are rather complex to model in this setting. This is because these contracts dictate that an entity periodically send messages, while the very notion of periodicity can only be understood over an interval of time. That is, periodic broadcasts are those that happen at certain time intervals. It does not make sense to say of a state that periodic broadcast is occurring at that state.

Well, then, clearly to interpret periodic broadcast, and in turn contracts with transfer, it is necessary to look at richer structures than merely states. An initial thought is to use runs through the state-space. That is, a sequence of states, possibly infinite but countable, can be drawn from the set of all possible states with valid behavioural transitions linking successive states according to a behavioural model of the form introduced in Section 3.3.2. A contract can then be deemed to either hold or not hold over such a sequence. For example, if the contract stipulated periodic message broadcast, then any run exhibiting this behaviour would uphold the contract, whereas other runs would violate it.

Of course, any particular state will consist of multiple entities. Each entity will have its own individual contract to obey. Hence it is more correct to say that a run adheres to an assignment of contracts if all of the entities to which the corresponding contracts are assigned exhibit the appropriate behaviours over the course of the run. Verification would then entail that all runs obeying an assignment are safe. That is, every state along each such run is safe according to the safety constraint.

A model of responsibility via runs and contracts seems reasonable. However, an alternative approach is devised for the revised model that is arguably superior. Instead of specifying contracts via behaviours over entire runs, each entity can be equipped with a specific on-board program (protocol) that is intended to control the entity's behaviour so that it remains responsible. The behaviour of this program is then separately defined at the level of the programming language used to define it. That is, the programming language is given a semantics so that every term in the language has a precise behaviour.

A notable advantage to this approach is that it provides a separation of concerns between the definition of a behavioural model and the specification of responsibility. That is, the programming language semantics are defined in a general sense and incorporated into the behavioural model that is defined over the state-space. The specification of the program for implementing responsibility can then be done separately, and can be easily changed without changing the semantics, which are defined for the general programming language. Another way of looking at this is that once the behavioural model of the "hardware" of entities and the software language are defined, entities become "programmable" by simply choosing software processes to control them.

Another advantage of the use of a protocol is that it is closer to a real-world implementation. While a contract is a set of conditions on the behaviour of an entity, the specifics of how these conditions are to be realised are not included in the definition of the contract. Specifying a protocol in an abstract programming language is closer to how a real system would be implemented.

Recall from the specification of the revised contract with transfer in Section 3.3.4 the condition of entities starting in fail-safe modes. This cannot be enforced by the protocol. In fact, the very notion of "start state" is as yet ambiguous. While it is clarified what is meant by this in later chapters, it suffices here to say that to capture this condition the notion of an initial state is defined. These are the states in which all entities are in fail-safe modes and are running the protocol as their on-board software controllers. The remaining conditions of the contract are embodied in the protocol term, which is interpreted behaviourally in terms of separately defined semantics for the language in which it is written.

Chapter 4

Language

In order to model the Comhordú protocol formally, some specification language is needed. Let us consider the domain we are attempting to model in order to reveal certain properties that would be desirable in such a language. The domain in question is that of systems of real-time mobile entities communicating over a wireless network. This immediately suggests that our language should have the following

- Entities and Networks: There should be terms that describe entities and networks of entities.
- Time: Intrinsic to Comhordú is the notion of real time, so any modelling language suited to Comhordú will have some model of time incorporated.
- Mobility: This refers to spatial mobility; we would like our language to be able to describe entities which can move in space. For this to be the case, an entity in the language would need some notion of its position. There would also need to be some semantics of the language governing how this position can change with time, i.e. there would need to be some semantic model of movement of entities.
- Communication: Inter-entity communication within a network of entities should be possible. The type of communication should be broadcast i.e. one-to-many, since this is the type of communication achieved over wireless networks. However, collisions need not be modelled since these are covered of by an underlying model that is discussed below, called the Space-Elastic Model.

Now, let us consider the Comhordú protocol itself and extract further features desirable of our language. Two key aspects of the protocol should be fundamental to our language:

- Space-Elastic Model (SEM): Beneath the Comhordú protocol, it is assumed that there is a lower level model called the Space-Elastic Model (SEM) [Hughes, 2006]. The model is an abstract interface guaranteeing timely message delivery to a variable zone as well as bounded notification of the message delivery zone after message delivery. This rigid timeliness comes at the cost of space-elasticity, i.e. the *zone* to which messages are sent varies over time, depending on wireless coverage and other factors. The verification of SEM is beyond the scope of this work. Also, its implementation details are abstracted here, though there does exist an implementation of SEM called the Space Elastic Adaptive Routing protocol SEAR, also given in [Hughes, 2006]. This is discussed further in Section 4.2.
- Arbitrary Coverage: The coverage of a message is the area to which it was delivered in a timely fashion. As per SEM, this coverage is elastic i.e. variable over time. Since this variation depends on unknown environmental factors, which are for all intents and purposes random, a model of coverage in the language should be arbitrary i.e. non-deterministic.
- Modes: Comhordú applies to systems in which every entity is at all times in some mode of operation, referred to as its mode. Over time, entities can transition from one mode to another. Terms in the language representing entities should incorporate a mode. To facilitate reasoning with modes, which is

needed for the verification of Comhordú, the mode should be clear from the syntax and easy to extract from an entity term. Mode transitions should be modelled, i.e. there should be some means for an entity term to evolve to another term with a different mode.

The question now arises as to what kind of formalism is suitable for modelling such systems? A language is needed that can describe entities and networks thereof with the above properties. To achieve this, we first focus on the model of a single entity, then entity networks are easily obtained via some sort of parallel composition operator.

The approach taken here is a many-tiered approach inspired by contemporary work in the area e.g. [Merro, 2009, Merro et al., 2011]. The idea of a many-tiered approach is that a language consists of two or more layers. For example, in [Merro et al., 2011], the calculus CWS is split into a "process layer" and a "network layer". Separate semantics are given for each, but the network-layer semantics subsumes those at the process-layer. In other words, first a process language is presented with its own semantics, then a network language is defined in terms of this process language. Arguably, any process language that relies on an underlying expression language could be referred to as a two-tiered language, though usually the expression language is not considered important enough to warrant such a consideration.

The language chosen to model Comhordú is referred to here as "many-tiered". Rather than one outer language and one inner language, there is one outer language and a number of inner languages. Perhaps this should still be considered a "two-tiered" language, since there are still only two levels in the language hierarchy, underlying expression languages aside. However the classification of this language as one or the other seems an unimportant detail, at least in this context.

Now, the question may arise as to why there are so many separate languages? Well, to briefly revisit the properties that the language should have, there is: broadcast, time, modes, an underling SEAR protocol, movement. Furthermore, there must be some language constructs in which the protocol itself can be expressed, as opposed to the physical dynamics of the systems. This plethora of attributes is difficult to find/model in a single language without a large amount of "hacks", leading to complex, verbose terms. In contrast, by modelling many features directly into the language, at the expense of complicating the language, the specification of terms therein becomes more straightforward. This is reminiscent of a standard trade-off between software and hardware. Since one of the key aims of this work is to provide a proof of safety, elegance of the language terms seems more important than minimalism in the language rules, hence the adoption of the many-tiered approach.

To model a single entity we choose to syntactically separate in our language those components of an entity which are conceptually separate i.e. its software controller, its mode and the interface to its underlying SEAR protocol. These languages are all given point-to-point semantics a.k.a. handshake communication semantics. The "outer" entity language then builds upon these languages, but its communication primitive is broadcast i.e. entities communicate to other entities via one-to-many communication, while internally point-to-point communications occur between the various sub-components. Figure 4.1 depicts an entity, its components and how they are connected via specially named channels for point-to-point communication. In the diagram, **P** is the software process controlling the entity, **I** is the interface to SEM, **K** is the component keeping track of the mode, and l is the current position of the entity. A number of curved lines represent the broadcast between this entity and its environment i.e. other entities in the network; this communication is bi-directional i.e. the entity sends and receives broadcast messages to/from its environment.

A further aim of splitting the language the way it has been done is to encode the hardware and "firmware" directly into the operational semantics, while leaving the software controller of an entity to be chosen by the system designer. This makes a clear-cut distinction between what is assumed to pre-exist and what Comhordú adds. It also allows the software controller to be easily replaced by another without having to change the rest of the language; a replacement software process could either improve on the existing one in some way or could even provide a different set of behaviours. Note the dotted line in Figure 4.1 indicating the that the software component is not hard-coded into the language, it is what must be designed here.

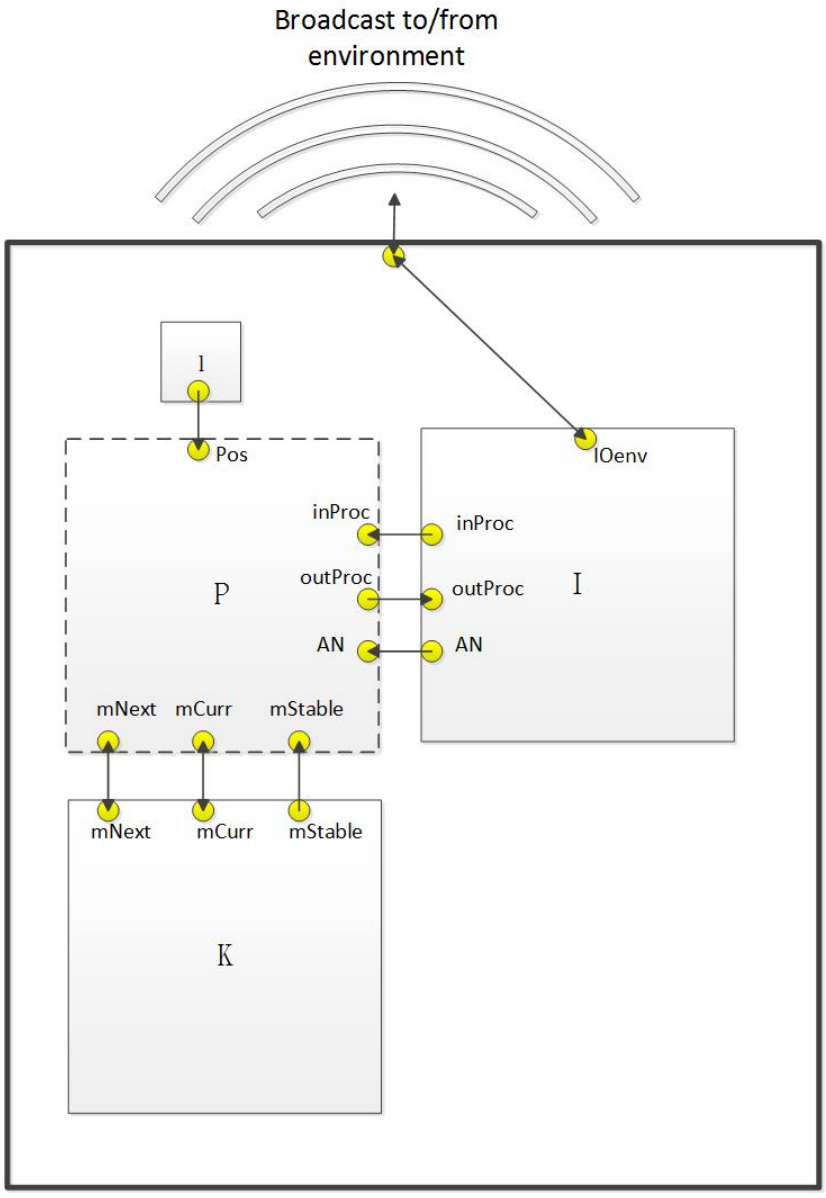


Figure 4.1: Components of an entity with distinguished channels for communication. Note that the component l is simply a position vector and does not have any associated language or SOS rules.

All the ingredients have now been gathered to build a layered process calculus for systems of mobile entities. The language is defined in terms of three sub-languages and one outer language. The sub-languages are the software language, the interface language and the mode state language. The software language is essentially timed CCS [Yi, 1991] with parametrisation and value passing and is presented in Section 4.1. The interface language, which models an interface to SEAR, follows in Section 4.2. The mode-state language, which models the mode-transition hardware/firmware of the entity is given in Section 4.3. There is no need to include an entire language for positions since the semantics governing the change of position are quite simple. The top-level entity language ties the three lower level languages together by linking, hiding and propagating the appropriate actions of the sub-components to produce new actions at the entity level; this is presented in Section 4.4.

4.1 Software Language

In this section, a process calculus for the software component of an entity is presented. This software calculus represents a programming language and hence should be flexible i.e. a programmer should be able to write processes for arbitrary purposes using this language. The calculus, by virtue of being used to control an entity in real time, should include some notion of time. Finally, since entities in the domain under consideration send messages to one another, there should be some message-passing primitives in the language. The language chosen, which has all of the aforementioned properties, is a standard CCS style value passing language much like the language found in [Bergstra et al., 2001b], with the slightly less standard addition of a delay prefix and delay transition semantics as found in the Timed CCS (TCCS) [Yi, 1991]. This language is built upon a set of base types and a Boolean expression language, neither of which is defined here.

4.1.1 Naming Conventions

In the following, a naming convention is adopted in which various symbols and letters are taken to range over specific types. The advantage of this is that explicit typing is then avoided in future. Some of these symbols may be referred to as "meta-variables". However the use of this term seems overly formal; questions are then raised such as what is object language(s) and what is the meta language? Rather, these symbols may simply be interpreted as arbitrary members of certain types, with the same symbols ranging over the same types for notational convenience. The presentation here is somewhat tedious, simply listing off various symbols and associating them with certain types. In light of this, the reader may skip over this section and return to it if any notational ambiguity arises later.

Let us now begin with the symbol v; this is assumed to range over the base type, upon which the language is parametrised. The identifier x represents some variable over the base type. Mixing these two, u is either a value or a variable of the base type. We use A and B to denote arbitrary types. Variables and values may be typed in certain contexts; for example $x : A$ restricts the variable x with the type A. Similarly, when we write $v : A$, we restrict the type of a value. Vectors of values, variables or both are denoted by placing an arrow over the appropriate symbol. For example $\vec{u}$ is a vector of variables/values, a vector being simply a comma delimited list. For typed vectors of values we write $\vec{v} : \vec{A}$, which is shorthand for

$$v_1 : A_1, v_2 : A_2, \ldots, v_n : A_n$$

where n is the vector length. Along the same lines we write $\vec{x}$ and $\vec{u} : \vec{A}$ for typed vectors of variables or typed vectors of variables/values.

Arbitrary expressions are denoted by e. These belong to a simple underlying expression language, the details of which are unimportant here other than the fact that closed expressions may evaluate to values of the base type. Not all expressions are closed: some may contain variables. These cannot evaluate to anything. The details of the evaluation function are too low level for inclusion here; furthermore the idea of an uninterpreted

expression language and evaluation function makes this language more flexible and general. Along with single expressions are vectors of expressions, represented by $\vec{e}$. Expression evaluation is denoted by

$$[\![e]\!] = v$$

This says that e evaluates to v. Evaluation can also apply to lists of expressions, yielding lists of base values

$$[\![\vec{e}]\!] = \vec{v}$$

Note that evaluation is a partial function; it is only defined on closed inputs i.e. expressions without any free variables.

To represent a channel we use c; channels are a foundational type upon which the language is built. Delays are represented by d; these are strictly positive, i.e. non-zero, real numbers. Ranging over time is t, a non-negative real number differing only from d in that it may be zero i.e and represents a time prefix as opposed to a delay action. For discrete actions we use δ; this is any action that is not a delay, and is defined by

$$\delta ::= c\langle\vec{v}\rangle! \mid c\langle\vec{v}\rangle? \mid \tau$$

The symbol α represents any action, either a discrete action or a delay:

$$\alpha ::= \delta \mid d$$

The tags ! and ? mark an action as either output or input respectively. Processes are denoted by **P** and **Q**. Primes are usually used to mark one process as the derivative of another; $\mathbf{P}'$ and $\mathbf{Q}'$, read as "**P** primed" and "**Q** primed" are derivative[1] processes of **P** and **Q** respectively. $\mathbf{P}''$ is also used, usually as a derivative of $\mathbf{P}'$, and so on. Terms **T** are similar to processes but are allowed to have free variables. These naming conventions are summarised in Table 4.1.

Table 4.1: Naming convention legend, establishing the meaning of certain expressions.

Expression	Explanation
v	A value of one of the base types.
x	A variable.
u	Either a variable or a base value.
e	An expression i.e. term in the underlying expression language.
$\vec{e}$	A vector of expressions.
A, B	Types
$v : A$	Typing example: a typed value.
$\vec{x}$, $\vec{v}$, $\vec{u}$	Vectors of variables, values, or both mixed.
$\vec{x}$	Example of a typed vector.
c	A channel.
d	Delay: non-zero real number.
t	Time value: a non-negative real number.
δ	A discrete action i.e. not a delay.
α	Any action, discrete or otherwise.
T, $\mathbf{T}'$	Open terms.
?, !	Input and output tags respectively.

Continued on Next Page...

[1] A derivative process $\mathbf{P}'$ of another process **P** is one such that there is some action α with $\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'$.

Table 4.1 – Continued

Expression	Explanation
P, **Q**	Processes- closed terms.
$\mathbf{P}'$, $\mathbf{P}''$, **Q**	Derivative processes.

4.1.2 Syntax of the Software Language

Table 4.2 presents the syntax of the software language. Most of the syntax should be familiar to anyone versed in value passing CCS, with the possible exception of the delay operator. For further reading on this, see [Bergstra et al., 2001b]. Note the use of the symbol **T** as opposed to **P** when defining the syntax. The idea here is that **T** is an open term, with possibly free variables, while **P** is a closed term, or a process. While the syntax defines a set of terms, the semantics applies only to processes, i.e. open terms are not included. However in the following, we discuss the terms of the syntax informally as if they were processes, ignoring the fact that they may have free variables. The aim here is not to be completely formal, but to get an idea of what the terms represent. The reason for having terms in the first place will become clear when we consider variable binding on input actions. Let us now discuss each term in the syntax and explain informally what it means.

- **0** : The nil process. This process should be capable of no discrete actions i.e. input, output or internal computation. The only thing the nil process should be allowed to do is idle indefinitely, i.e. let any amount of time pass.
- $c\langle\vec{e}\rangle!.\mathbf{T}$: This term is ready to output the message $\vec{e}$ over the channel c and then evolve to become **T**. Note that $\vec{e}$ is a vector of expressions and must evaluate to some vector of values if the process is to perform an output.
- $c\langle\vec{x}\rangle?.\mathbf{T}$: This term is listening for an incoming message. The message it listens for is a vector of values. The variables in the vector $\vec{x}$ are bound in the term **T**. When a specific vector of values is inputted, the process evolves to **T** with all free occurrences of those variables replaced by the received values. Notice that the variables of the input prefix are typed. This allows us to define the semantics such that only vectors of values matching the types of these variables may be inputted.
- **if** b_{exp} **then T** : The behaviour of this term depends on the value of the Boolean guard b_{exp}. If the guard evaluates to true, then the term behaves like **T**. Otherwise, it behaves like the nil process, i.e. it can do nothing but idle indefinitely.
- $\mathbf{P}_1 + \mathbf{P}_2$: The choice construct. Let's say we have a set of indexed terms $\mathbf{T}_i$ each indexed by an element i of some set I. Then we can "sum up" all these terms to form a choice term. The resulting term is then willing to perform any discrete action that one of its sub-terms can perform, and subsequently evolve to the corresponding derivative of that term. The choice term can also delay if *all* of its sub-terms can delay. The derivative of a choice term after a delay is the choice term formed by "summing up" all the individual derivatives of the sub-terms. Note that while discrete actions "collapse" choice, so to speak, delays preserve it.
- $\mathbf{T} \mid \mathbf{T}$: Parallel composition. This construct models two terms executing concurrently. If either sub-term can do a discrete action and evolve to some derivative, then this can do the same action, and said sub-term in question is replaced by its derivative in the derivative of the entire parallel term. Also, if both sub-terms can do complementary discrete actions, the overall term can do an internal action τ and evolve to a parallel construct where each sub-term is replaced by its derivative. Finally, if both sub-terms can delay, then so can the top level term. Note that all actions preserve the parallel structure.
- $\mathbf{N}(\vec{u})$: Application of a parametrised term to values. Parametrised terms $\mathbf{N}(\vec{u})$ are given associated definitions

$$\mathbf{N}(\vec{x}) \stackrel{\text{def}}{=} \mathbf{T}$$

where **T** is some other term and **N**is drawn from a set of names. Such a term behaves exactly like its definition body with all free occurrences of $\vec{x}$ replaced by all the values of $\vec{u}$. This feature allow for infinite or looping behaviour in the language. Notice that in a process definition, the parameters are typed. We stipulate that such definitions are only valid if they are well typed i.e. if the term **T** uses the variables in $\vec{x}$ according to how they are typed.

- $\varepsilon(t)\bullet\mathbf{T}$: The delay prefix. This term can be thought of as being put to "sleep" for a time specified by t. In the special case where t is 0, the term behaves exactly as **T**. Otherwise, it must allow t units of time to pass, i.e. it must delay to a derivative process, before the actions of **T** become available to perform. When this process delays by an amount d less than or equal to the guard t, the derivative process is that with the delay subtracted from the guard i.e. $\varepsilon(t-d)\bullet\mathbf{T}$.

Table 4.2: Syntax of the software language.

$$\mathbf{T} ::= \mathbf{0} \mid c\langle\vec{e}\rangle!\bullet\mathbf{T} \mid c\langle\vec{x}\rangle?\bullet\mathbf{T} \mid \textbf{if } b_{exp} \textbf{ then } \mathbf{T} \mid \mathbf{T}_1 + \mathbf{T}_2 \mid \mathbf{T} \mid \mathbf{T} \mid \mathbf{N}(\vec{e}) \mid \varepsilon(e)\bullet\mathbf{T}$$

Remark 4.1 (Typing)**.** The typed vectors $\vec{x}$ appearing in input prefixed terms and parametrised process definitions suggest that there is some notion of typing in this language. Indeed, but it is not discussed at length here. There is assumed to be an underlying set of well-typing rules for the Boolean expression language. These rules can be lifted to the language of process terms by the simple observation that computations on values are only ever performed within Boolean expressions. No other constructs in the language inspect values or use them in any way other than to pass them on, hence well-typing is an issue only at the level of Boolean expressions. Just as the Boolean expression language is left wholly undefined in this section, so will these well typing rules, though they will be assumed to exist and are hence a necessary feature in any Boolean expression language underlying this one. What will be assumed here is that the semantic transition relations are defined only over well-typed processes. ▲

4.1.3 Discrete Software Language Semantics

Table 4.3 presents the semantics of discrete actions as a labelled transition system. Judgements in these semantics will be of the form

$$\mathbf{P} \xrightarrow{\delta} \mathbf{P}'$$

which may be read as "**P** transitions to $\mathbf{P}'$ by δ." Recall from Section 4.1.1 that a discrete action δ is either the input/output of any value or the internal action τ. These are the actions that should be familiar from value passing CCS as presented in [Bergstra et al., 2001b]. The laws governing these actions should also be familiar. Hence anybody comfortable with value passing CCS can probably skip straight to the timed laws of Table 4.5, discussed in Section 4.1.5, which are slightly less standard. Let us now explain these laws using examples. It is assumed for these examples that the natural numbers $\mathbb{N}$ constitute one of the base types.

The first three rules deal with the case where an action prefixed process performs the action appearing in the prefix and evolves to the process directly after the prefix. PFX-TAU and PFX-OUT are straightforward. PFX-IN-MSG warrants further explanation due to the fact that it deals with variable binding. The premise in this law, that is $|\vec{v}| = |\vec{x}|$, asserts that the vector of values $\vec{v}$ is of the same length as the vector of variables in the input prefix construction. The conclusion of this law deals with the substitution of free variables in **T** with the values that have been input, i.e. $\vec{v}$. Each component v_i replaces x_i wherever it occurs free in **T**, and the resulting process with all substitutions performed is the derivative process. For an example of this let us consider the process

$$chan1\langle x:\mathbb{N}, y:\mathbb{N}\rangle?\bullet chan2\langle x,y\rangle!\bullet\mathbf{0}$$

This process is willing to input any pair of natural numbers $\langle x, y \rangle$ on the channel *chan*1 and then output these numbers on *chan*2. By the rule PFX-IN-MSG, we can choose and pair of values as long as they are natural numbers and this process can consume them. Let's choose the pair $\langle 17, 19 \rangle$, which happen to be twin primes. Then we have by our input rule

$$chan1\langle x : \mathbb{N}, y : \mathbb{N} \rangle ?.chan2\langle x, y \rangle !.\mathbf{0} \xrightarrow{chan1\langle 17,19 \rangle ?} chan2\langle 17, 19 \rangle !.\mathbf{0}$$

Notice what has happened here. The bound variables x and y have been replaced by the received values 17 and 19 in the derivative process $chan2\langle 17, 19 \rangle !.\mathbf{0}$. We are now left with a process that is willing to output on the channel *chan*2 said pair of numbers and then become the nil process. Let us explicitly state this transition:

$$chan2\langle 17, 19 \rangle !.\mathbf{0} \xrightarrow{chan2\langle 17,19 \rangle !} \mathbf{0}$$

The remaining process $\mathbf{0}$ can no longer perform any discrete transitions.

The rule THEN-T handles the case of an if-then process whose guard is true. The rule essentially says that when the guard is true, all discrete actions of the then branch become available to the entire process. Let us assume that equality of two natural numbers is a term in our Boolean expression language. Then we may construct the process

$$\textbf{if } 14 = 14 \textbf{ then } chan2\langle 17, 19 \rangle !.\mathbf{0}$$

Assuming that our Boolean expression evaluation is working correctly, the expression in the guard, $14 = 14$, should evaluate to true i.e.

$$[\![14 = 14]\!] = \textbf{true}$$

where $[\![b_{exp}]\!]$ denotes the evaluation, if it exists, of the expression b_{exp}. This satisfies of the premises of THEN-T. We also have, from the previous example, that

$$chan2\langle 17, 19 \rangle !.\mathbf{0} \xrightarrow{chan2\langle 17,19 \rangle !} \mathbf{0}$$

which satisfies the other premise. Hence we can conclude by this rule that

$$\textbf{if } 14 = 14 \textbf{ then } chan2\langle 17, 19 \rangle !.\mathbf{0} \xrightarrow{chan2\langle 17,19 \rangle !} \mathbf{0}$$

Note that when the guard is false, a process is essentially the same as $\mathbf{0}$ i.e. it cannot perform any discrete action.

We now consider the choice construct and the rule by which it may perform an action. A choice construct between two processes is written $\mathbf{P}_1 + \mathbf{P}_2$. Sometimes more than two processes can appear in a "sum" of processes; formally speaking these constitute repeated applications of the choice constructor with either left or right associativity. Such a sum would be written as

$$\mathbf{P}_1 + \mathbf{P}_2 + \cdots + \mathbf{P}_n$$

In the following example, we consider a choice between two processes, written as

$$chan1\langle \text{“}Hello\text{”} \rangle ?.\mathbf{0} + chan2\langle 17, 19 \rangle !.\mathbf{0}$$

Now, it has already been established that the second process in the sum can perform the transition

$$chan2\langle 17, 19 \rangle !.\mathbf{0} \xrightarrow{chan2\langle 17,19 \rangle !} \mathbf{0}$$

and since this process is a component of the choice, the entire sum can perform this transition also

$$chan1\langle \text{“}Hello''\rangle?.\mathbf{0} + chan2\langle 17,19\rangle!.\mathbf{0} \xrightarrow{chan2\langle 17,19\rangle!} \mathbf{0}$$

Similarly, this sum process can perform any action its left hand component can perform i.e.

$$chan1\langle \text{“}Hello''\rangle?.\mathbf{0} + chan2\langle 17,19\rangle!.\mathbf{0} \xrightarrow{chan1\langle \text{“}Hello''\rangle?} \mathbf{0}$$

because

$$chan1\langle \text{“}Hello''\rangle?.\mathbf{0} \xrightarrow{chan1\langle \text{“}Hello''\rangle?} \mathbf{0}$$

by PFX-IN-MSG.

Table 4.3: Semantics of discrete actions in the software language, defined over processes, which are closed terms.

PFX-OUT	$\dfrac{[\![\vec{e}]\!] = \vec{v}}{c\langle\vec{e}\rangle!.\mathbf{P} \xrightarrow{c\langle\vec{v}\rangle!} \mathbf{P}}$
PFX-IN-MSG	$\dfrac{\lvert\vec{v}\rvert = \lvert\vec{x}\rvert}{c\langle\vec{x}\rangle?.\mathbf{T} \xrightarrow{c\langle\vec{v}\rangle?} \mathbf{T}[\vec{v}/\vec{x}]}$
THEN-T	$\dfrac{\mathbf{P} \xrightarrow{\delta} \mathbf{P}' \qquad [\![b_{exp}]\!] = \textbf{true}}{\textbf{if } b_{exp} \textbf{ then } \mathbf{P} \xrightarrow{\delta} \mathbf{P}'}$
CHO-DISC-L	$\dfrac{\mathbf{P}_1 \xrightarrow{\delta} \mathbf{P}'}{\mathbf{P}_1 + \mathbf{P}_2 \xrightarrow{\delta} \mathbf{P}'}$
CHO-DISC-R	$\dfrac{\mathbf{P}_2 \xrightarrow{\delta} \mathbf{P}'}{\mathbf{P}_1 + \mathbf{P}_2 \xrightarrow{\delta} \mathbf{P}'}$
PAR-DISC-L	$\dfrac{\mathbf{P} \xrightarrow{\delta} \mathbf{P}'}{\mathbf{P} \mid \mathbf{Q} \xrightarrow{\delta} \mathbf{P}' \mid \mathbf{Q}}$
PAR-DISC-R	$\dfrac{\mathbf{Q} \xrightarrow{\delta} \mathbf{Q}'}{\mathbf{P} \mid \mathbf{Q} \xrightarrow{\delta} \mathbf{P} \mid \mathbf{Q}'}$
SYNC-LR	$\dfrac{\mathbf{P} \xrightarrow{c\langle\vec{v}\rangle!} \mathbf{P}' \qquad \mathbf{Q} \xrightarrow{c\langle\vec{v}\rangle?} \mathbf{Q}'}{\mathbf{P} \mid \mathbf{Q} \xrightarrow{\tau} \mathbf{P}' \mid \mathbf{Q}'}$
SYNC-RL	$\dfrac{\mathbf{P} \xrightarrow{c\langle\vec{v}\rangle?} \mathbf{P}' \qquad \mathbf{Q} \xrightarrow{c\langle\vec{v}\rangle!} \mathbf{Q}'}{\mathbf{P} \mid \mathbf{Q} \xrightarrow{\tau} \mathbf{P}' \mid \mathbf{Q}'}$
APP	$\dfrac{\mathbf{N}(\vec{x}) \stackrel{\text{def}}{=} \mathbf{T} \qquad \mathbf{T}[\vec{e}/\vec{x}] \xrightarrow{\delta} \mathbf{P}'}{\mathbf{N}(\vec{e}) \xrightarrow{\delta} \mathbf{P}'}$
DEL-TOUT-DISC	$\dfrac{\mathbf{P} \xrightarrow{\delta} \mathbf{P}' \qquad [\![e]\!] = 0}{\varepsilon(e).\mathbf{P} \xrightarrow{\delta} \mathbf{P}'}$

The rules PAR-DISC-L and PAR-DISC-R allow processes in parallel to interleave their actions. That is, if a process on either side (left or right) of the operator can do an action, then this action is lifted to the entire parallel construct and the derivative of the sub-process replaces it in the overall parallel process. Let us take an

example demonstrating these interleaving rules. To do so we consider the process

$$chan1\langle x:\mathbb{N}, y:\mathbb{N}\rangle?.chan1\langle x,y\rangle!.\mathbf{0} \mid chan2\langle 17,19\rangle!.\mathbf{0}$$

Now, we know from previous examples that the right sub-process is capable of the following:

$$chan2\langle 17,19\rangle!.\mathbf{0} \xrightarrow{chan2\langle 17,19\rangle!} \mathbf{0}$$

Hence, by the rule PAR-DISC-R we can lift this transition to the parallel structure, yielding the transition:

$$chan1\langle x:\mathbb{N}, y:\mathbb{N}\rangle?.chan1\langle x,y\rangle!.\mathbf{0} \mid chan2\langle 17,19\rangle!.\mathbf{0} \xrightarrow{chan2\langle 17,19\rangle!} chan1\langle x:\mathbb{N}, y:\mathbb{N}\rangle?.chan1\langle x,y\rangle!.\mathbf{0} \mid \mathbf{0}$$

This derivative process can further evolve via PAR-DISC-L as follows:

$$chan1\langle x:\mathbb{N}, y:\mathbb{N}\rangle?.chan1\langle x,y\rangle!.\mathbf{0} \mid \mathbf{0} \xrightarrow{chan1\langle 1,10\rangle?} chan1\langle 1,10\rangle!.\mathbf{0} \mid \mathbf{0}$$

Note that the input rule was used implicitly here. Finally, we can perform one more discrete action:

$$chan1\langle 1,10\rangle!.\mathbf{0} \mid \mathbf{0} \xrightarrow{chan1\langle 1,10\rangle!} \mathbf{0} \mid \mathbf{0}$$

The process $\mathbf{0} \mid \mathbf{0}$ is then incapable of any further discrete transitions and is hence behaviourally similar to $\mathbf{0}$.

The parallel construct is capable of more than just interleaving actions. Actions may be synchronised between its sub-components i.e. one component may perform an output on a particular channel while the other may perform an input on that same channel. From an external point of view, this synchronisation is an internal computation and hence the top-level process is observed to perform a silent action τ. The rules SYNC-LR and SYNC-RL capture the behaviour of message passing from left to right or right to left respectively between the sub-processes of a parallel construct. We provide an example demonstrating the rule SYNC-RL; the other rule is symmetrically similar and does not warrant a separate example. The process we choose is:

$$chan1\langle x:\mathbb{N}, y:\mathbb{N}\rangle?.chan2\langle x,y\rangle!.\mathbf{0} \mid chan1\langle 17,19\rangle!.\mathbf{0}$$

Note that the left and right sub-components of this process are capable of performing complementary actions i.e. they are prefixed by the same channel, one is input while the other is output, and the values being output match the type of the input vector. Since the right process is willing to output a pair of numbers and the left process is willing to input these numbers, by the rule SYNC-RL we have:

$$chan1\langle x:\mathbb{N}, y:\mathbb{N}\rangle?.chan2\langle x,y\rangle!.\mathbf{0} \mid chan1\langle 17,19\rangle!.\mathbf{0} \xrightarrow{\tau} chan2\langle 17,19\rangle!.\mathbf{0} \mid \mathbf{0}$$

Note what has happened here: the right sub-process has essentially sent the pair of values 17 and 19 over to the left process, and the left sub-process has consumed these values and evolved to a process ready to output them on *chan*2, namely

$$chan2\langle 17,19\rangle!.\mathbf{0}$$

Let us ignore the rule DEL-TOUT-DISC for the time being, it is discussed later along with a discussion of the delay prefix. The final rule in this set of semantics that we discuss then is APP. This rule deals with the application of a parametrised process definition to a vector of values. Let us begin first with what a parametrised process definition is. Given a vector of variables $\vec{x}$, some name $\mathbf{N}$, and a term $\mathbf{T}$, one can construct a parametrised process definition

$$\mathbf{N}(\vec{x}) \stackrel{\text{def}}{=} \mathbf{T}$$

Now, as long as this definition is well typed, $\mathbf{N}(\vec{u})$ becomes a term of the language whose behaviour is exactly that of $\mathbf{T}$ with appropriate substitutions made. This is captured by the rule APP. The first premise makes the

connection between the name **N** and its body **T** in the definition. The second premise of the rule asserts that the term **T** with all free occurrences of variables in $\vec{x}$ replaced by corresponding values in $\vec{v}$ can do an action. Implicit in this assertion is that the term after the substitution, i.e. $\mathbf{T}[\vec{v}/\vec{x}]$, is in fact a process, with no remaining free variables once those in $\vec{x}$ have been replaced by values.

The conclusion of this law states that the application $\mathbf{N}(\vec{v})$ of a parametrised process to a vector of values can do the same action as can its body in the definition. To demonstrate the rule APP we take an example. Let's say that the following definition has been made:

$$repOutPair(x,y) \stackrel{\text{def}}{=} chan1\langle x,y\rangle!.repOutPair(x,y)$$

Let's consider instantiating the parameters of this term to concrete values in order to make it a closed term, for example

$$repOutPair(11,13)$$

Then since we are assuming this definition, and since by the rule for output

$$chan1\langle 11,13\rangle!.repOutPair(11,13) \xrightarrow{chan1\langle 11,13\rangle!} repOutPair(11,13)$$

we have satisfied both premises for our rule APP. Hence we can make the conclusion that

$$repOutPair(11,13) \xrightarrow{chan1\langle 11,13\rangle!} repOutPair(11,13)$$

It is evident that this process can run ad infinitum outputting the same pair of values over and over again. Clearly, parametrised process definitions inject a lot of power into this language by allowing processes with infinite behaviour such as this one.

Remark 4.2 (Guardedness)**.** One thing to notice about the process in the explanation of the rule APP is that it is defined in terms of itself i.e. the process name appears in the body of its own definition. In general, process names in the body of such definitions should be guarded, i.e. they should appear as a sub-term of some term whose top level structure is an action prefix. In the example we are considering here, this is certainly the case: the only occurrence of *repOutPair* in the body of its own definition is guarded by the action prefix $chan1\langle x,y\rangle!.$. The restriction of guardedness upon process definitions is not completely necessary; however guarded terms are in a certain sense guaranteed to behave well, whereas unguarded terms may be nonsensical. For a simple example of why this is the case, consider some process

$$nonsense(x,y) \stackrel{\text{def}}{=} nonsense(x,y)$$

Since $nonsense(x,y)$ is a term in the syntax, it can constitute the body of a definition, in particular its own definition. However, this process can never perform any action due to the circular reasoning that it generates, i.e. "If $nonsense(x,y)$ can do an action, then $nonsense(x,y)$ can do that action". Notice that this process truly is stuck since the only rule applicable to an application structure is APP, and this is the very rule that generates said circular reasoning.

For the time being, such processes don't seem like too much trouble: if one is willing to define something so absurd, let it remain stuck. The intuition is that these processes will simply behave as the nil process **0** and will not affect any other processes with which they are composed. However, it will become clear when the timed semantics are presented that these processes cause trouble: they are both deadlocked from performing any discrete actions and time-locked, whereas the nil process does allow time to pass. Therefore careful attention should be taken by the "programmer" to avoid such definitions. ▲

4.1.4 Towards a Delay Semantics for Software Processes

The reason for splitting up the discrete and timed semantics is that the latter are slightly more involved, relying on some auxiliary machinery. This section motivates the need for such machinery and gives some auxiliary definitions that lay the groundwork for the delay semantics which follow. The main issue behind all of the complications of the delay semantics is a language property known as maximal progress. The discussion begins on this subject in Remark 4.3 and then moves towards auxiliary definitions.

Remark 4.3 (Maximal Progress)**.** According to [Yi, 1991], a process calculus is said to have maximal progress whenever the following is true: "If an agent can proceed by performing actions, then it should never wait unnecessarily." What is meant here by "Agent" is a term in the language. Note the implicit universal quantification over all terms of the language. Many papers refer to a "maximal progress assumption", which is the assumption that maximal progress is indeed a desirable language property. This assumption is often cited to justify certain choices in the operational semantics of the language, choices which intuitively should lead to maximal progress as an emergent property.

In [Yi, 1991], maximal progress is shown to emerge as the result of a number of choices in the structural operational semantics. Firstly, it is argued that the actions under consideration should only be internal actions a.k.a τ-actions. The justification for this is that these are the only actions over which a process has complete control. Input/Output actions on the other hand are sometimes referred to as half actions since they rely on environmental synchronisation. Thus they are not included in the interpretation of maximal progress for CCS. There are two ways in which internal actions may be performed by a CCS process. The first is simple: τ-prefixing. The second involves synchronisation between parallel components. In both cases, time should be frozen until the internal computation has halted: this is maximal progress for CCS. The following choices for the semantics are intended to ensure maximal progress.

- Not allowing τ-prefixed processes to delay. This is achieved by simply omitting a delay law for τ-prefixed processes.
- Enforcing a special side condition upon the delay law for parallel processes which ensures that whenever they delay, no synchronisation is possible between them at any point during the delay. This side condition is subtle, and is discussed in more detail below.

Let us explore an initial naive attempt to encode the special side condition for parallel processes, as is done in the textbook [Aceto, 2007]. The idea is to come up with some predicate *noSync*that captures the fact that two processes cannot synchronise across a certain time span. In particular, we would like to be able to add this predicate as a premise of the rule for parallel delay as in the following.

$$\frac{\mathit{noSync}(\mathbf{P},\mathbf{Q},d) \qquad \mathbf{P} \xrightarrow{d} \mathbf{P}' \qquad \mathbf{Q} \xrightarrow{d} \mathbf{Q}'}{\mathbf{P} \mid \mathbf{Q} \xrightarrow{d} \mathbf{P}' \mid \mathbf{Q}'}$$

This law says that if both components of the parallel construct can delay by some amount d, and no synchronisation is possible between them at any point during the delay, then the entire term can delay by the same amount, with the sub-terms evolving to their respective derivatives. The law appears straightforward at first glance, until one considers how the predicate *noSync* is formulated. Informally, the predicate should say that whenever the two processes in question can delay by some amount less than the given amount, the resulting processes cannot synchronise. More formally we define this predicate in Definition 4.4.

Definition 4.4 (No-synch Predicate for software processes)**.**

$$\begin{aligned}\mathit{noSync}(\mathbf{P},\mathbf{Q},d) \stackrel{\text{def}}{=} \mathit{noComm}(\mathbf{P},\mathbf{Q}) \wedge \\ \forall\, \mathbf{P}',\mathbf{Q}',d' < d, \\ \mathbf{P} \xrightarrow{d'} \mathbf{P}' \wedge \mathbf{Q} \xrightarrow{d'} \mathbf{Q}' \Rightarrow \mathit{noComm}(\mathbf{P}',\mathbf{Q}')\end{aligned}$$

■

where *noComm* is defined in Definition 4.5.

Definition 4.5 (No-comm Predicate for software processes)**.**

$$noComm(\mathbf{P},\mathbf{Q}) \stackrel{\text{def}}{=} \forall\, c, \vec{v},\\ \mathbf{P} \xrightarrow{c\langle\vec{v}\rangle?} \Rightarrow \neg\mathbf{Q} \xrightarrow{c\langle\vec{v}\rangle!} \wedge\\ \mathbf{P} \xrightarrow{c\langle\vec{v}\rangle!} \Rightarrow \neg\mathbf{Q} \xrightarrow{c\langle\vec{v}\rangle?}$$

■

While these formulae seem perfectly sound, they are problematic to work with due to the large amount of universal quantification involved. To prove a delay transition for a parallel composition of processes, one becomes obliged to prove for every delay less than this delay in question that no communication is possible at that point. This is an unwieldy proof obligation, and is not in keeping with the nature of the structural operational style in which the semantics should be built. Further to being difficult to work with on paper, this formulation completely fails to meet the requirements for an inductive definition in the calculus of constructions. This renders it an impossibility for inclusion in the Coq model. The problem is that occurrences of the delay relation in the premise of the law are hidden behind universal quantifiers, which is not allowed.

The question that then arises is whether there is another characterisation of this *noSync* predicate, one which is defined in terms of the structure of terms rather than one which relies on the semantics. Such a solution is indeed adopted in [Yi, 1991]; a function $Sort_d$ is defined over terms which gives, for each term, a coarse approximation of the actions available to that term within d units of time. The side condition for the parallel construct then states that there are no complementary actions available to the respective terms as per the function $Sort_d$. While this function only approximates the actions available to a process, it is enough to ensure the emergence of desirable language properties such as maximal progress.

For this language, the sort function of [Yi, 1991] is modified to account for value passing, parametrised definitions and conditional terms. This is defined in Definition 4.6. It is given there as a relation rather than a function because a relational characterisation is in fact more convenient to work with in Coq. To elaborate on this, rather than defining a set of actions for every process, we relate actions to processes, such that the totality of actions related to a particular process constitutes exactly the desired set, without it having to be explicitly constructed. ▲

Definition 4.6 (The *Sort* relation)**.** For a delay d, a discrete action δ and a process $\mathbf{P}$, the element of the sort relation $\delta\ Sort_d\ \mathbf{P}$ approximates whether or not the action δ can be performed within (but not including) d units of time by a delay derivative of $\mathbf{P}$. The exact meaning of this relation is hard to defined because it is used to define the very delay relation it hopes to approximate. Given in Table 4.4 are the rules for this relation. Notice the terms $|\vec{x}|$ and $|\vec{v}|$ appearing in one of the rules; these denote the lengths of the vectors $\vec{x}$ and $\vec{v}$ respectively.

The rules do not need much explaining, since they are quite similar to the discrete semantics, so they are briefly described here. The rules for input and output prefixed processes say that the action corresponding to the prefix is related to the process. The rule for the parallel and choice constructs, lifts the relation from a sub-term to the overall term. The rule for application processes lifts the sort relation from a process body to an application process. The only mildly interesting rule here is the delay prefix rule. This says that an instance of *Sort* with parameter d in the guarded sub-process implies an instance in the overall process with parameter $t+d$. The intuition behind adding t to the time parameter is that this time must elapse before the delay guard times out and the actions of the guarded sub-process are "unlocked" so to speak. This will become more clear when the delay semantics are presented in Table 4.5.

Table 4.4: Inductive rules for the *Sort* relation.

$\dfrac{[\![\vec{e}]\!] = \vec{v}}{c\langle\vec{v}\rangle!\ Sort_d\ c\langle\vec{e}\rangle!.\mathbf{P}}$	$\dfrac{\lvert\vec{x}\rvert = \lvert\vec{v}\rvert}{c\langle\vec{v}\rangle?\ Sort_d\ c\langle\vec{x}\rangle?.\mathbf{P}}$
$\dfrac{[\![e]\!] = t \qquad \delta\ Sort_d\ \mathbf{P}}{\delta\ Sort_{t+d}\ \varepsilon(e).\mathbf{P}}$	$\dfrac{[\![b_{exp}]\!] = \mathbf{true} \qquad \delta\ Sort_d\ \mathbf{P}}{\delta\ Sort_d\ \mathbf{if}\ b_{exp}\ \mathbf{then}\ \mathbf{P}}$
$\dfrac{\delta\ Sort_d\ \mathbf{P}_1}{\delta\ Sort_d\ \mathbf{P}_1 + \mathbf{P}_2}$	$\dfrac{\delta\ Sort_d\ \mathbf{P}_2}{\delta\ Sort_d\ \mathbf{P}_1 + \mathbf{P}_2}$
$\dfrac{\delta\ Sort_d\ \mathbf{P}_1}{\delta\ Sort_d\ \mathbf{P}_1 \mid \mathbf{P}_2}$	$\dfrac{\delta\ Sort_d\ \mathbf{P}_2}{\delta\ Sort_d\ \mathbf{P}_1 \mid \mathbf{P}_2}$
$\dfrac{\mathbf{N}(\vec{x}) \stackrel{\text{def}}{=} \mathbf{T} \qquad \delta\ Sort_d\ \mathbf{T}[\vec{e}/\vec{x}]}{\delta\ Sort_d\ \mathbf{N}(\vec{e})}$	

■

Definition 4.7 (The relation *noSyncSort*)**.** Building upon the definition of *Sort*, it is possible to formulate an alternative predicate to *noSync*without the problematic reference to the delay semantics. This new predicate is called *noSyncSort*. Recall that the aim of such a predicate is to ensure maximal progress in the language; hence rather than show that this new predicate is equivalent to *noSync*it suffices to show that maximal progress holds for the language, which is done in Section 4.1.7. In fact, the very question of whether the two predicates are equivalent is somewhat complicated: the predicate *noSync*relies on the delay semantics while *noSyncSort* is used to define those semantics.

Below is the definition of *noSyncSort*. The parameters are two processes **P** and **Q** and a delay d. It says that for all discrete actions δ, whenever that discrete action is related by *Sort* to the process **P**, then it is not the case that the complement of that action is related to **Q**. The complement of an action δ is denoted by $\overline{\delta}$.

$$noSyncSort(\mathbf{P},\mathbf{Q},d) \stackrel{\text{def}}{=} \forall\delta, \delta\ Sort_d\ \mathbf{P} \Rightarrow \neg\overline{\delta}\ Sort_d\ \mathbf{Q}$$

■

4.1.5 Software Language Timed Semantics

Now that the auxiliary material of the previous section has been covered, it is possible to introduce delay behaviour to the software language. Table 4.5 presents the timed transition rules for this language. Timed transitions are given via a separate set of semantics because they behave slightly differently to their discrete counterparts; for example, they do not collapse a choice construct. Judgements in the semantics of timed transitions are of the form

$$\mathbf{P} \xrightarrow{d} \mathbf{P}'$$

which may be read as "The process **P** delays to **P′** by d." The labels d on timed transitions are strictly positive real numbers and are called delays. Choosing the reals as the time domain facilitates arbitrary granularity in the time delays i.e. there is no bound on how small a time delay can be. Also, it seems more faithful to reality than a model based on, say, rational numbers. The restriction to non-zero reals is not overly important but does help guarantee certain language properties such as time determinacy, see Section 4.1.7.

Let us now discuss each of these timed transition rules, using examples where necessary to aid in their explanation. First for discussion are the rules for delay prefixed processes, since these are at the core of the timed semantics. The rule DEL-SUB says that a delay prefixed process may delay by any amount of time that

is less than or equal to its guard and evolve to a derivative where the value of the delay is subtracted from the guard. For example the process

$$\varepsilon(34).chan1\langle x\rangle?.chan2\langle x\rangle!.\mathbf{0}$$

is capable of performing any delay less than or equal to 34. The following transition is one such possibility:

$$\varepsilon(34).chan1\langle x\rangle?.chan2\langle x\rangle!.\mathbf{0} \xrightarrow{14} \varepsilon(20).chan1\langle x\rangle?.chan2\langle x\rangle!.\mathbf{0}$$

Note that the body remains the same, only the guard changes. When the value at the delay prefix reaches zero, we say that the delay has timed out, and the process behind the delay becomes activated i.e. all its transitions become available. This is precisely what the rule DEL-TOUT-DISC from the discrete semantics says. The final rule that involves the delay prefix operator is DEL-ADD. What this rule says is that delays can be added together i.e. if some process is capable of performing a delay, and it is then prefixed by another delay, the resultant process may perform the sum of both delays.

Observe Figure 4.2 and Figure 4.3 which help convey the intuitions behind the add/subtract delay laws respectively. These show a delay prefixed process performing a delay and evolving to a new process. Time is taken to move horizontally, flowing from left to right. Above the process transition in each case are a number of measurements to depict the lengths of the delays and their relationships.

For an example of these rules in action, let's take the process

$$\varepsilon(10).\varepsilon(11).chan\langle 532\rangle!.\mathbf{0}$$

By the subtraction law for prefixes, the inner process

$$\varepsilon(11).chan\langle 532\rangle!.\mathbf{0}$$

is capable of timing out after a delay of 11:

$$\varepsilon(11).chan\langle 532\rangle!.\mathbf{0} \xrightarrow{11} \varepsilon(0).chan\langle 532\rangle!.\mathbf{0}$$

Then, by the addition law, the overall process can time out after a delay of 10 + 11 = 21, i.e.

$$\varepsilon(10).\varepsilon(11).chan\langle 532\rangle!.\mathbf{0} \xrightarrow{21} \varepsilon(0).chan\langle 532\rangle!.\mathbf{0}$$

Since by previously discussed rules we know that

$$chan\langle 532\rangle!.\mathbf{0} \xrightarrow{chan\langle 532\rangle!} \mathbf{0}$$

then by the timeout rule we have that

$$\varepsilon(0).chan\langle 532\rangle!.\mathbf{0} \xrightarrow{chan\langle 532\rangle!} \mathbf{0}$$

The rules NIL, PFX-OUT-DEL and PFX-IN-DEL all apply to terms which are willing to idle indefinitely without changing. NIL says that the nil process can idle for any amount of time whatsoever; this is the only action that **0** is capable of, and is necessary so that when **0** is put into a choice or parallel context it does not create a timelock. The rules PFX-OUT-DEL and PFX-IN-DEL allow input and output prefixed processes to delay by an arbitrary amount of time without changing i.e. the derivative process is the same as the original. It is necessary to allow a process to delay when it is ready to perform an I/O action since there may be another process running concurrently that will soon be able to perform the complementary I/O action. In other words, if a process is waiting to synchronise, then it should let time pass while it does so. Notice that there is no similar rule for a process prefixed by the silent action τ. The omission of such a rule is a necessary step towards a

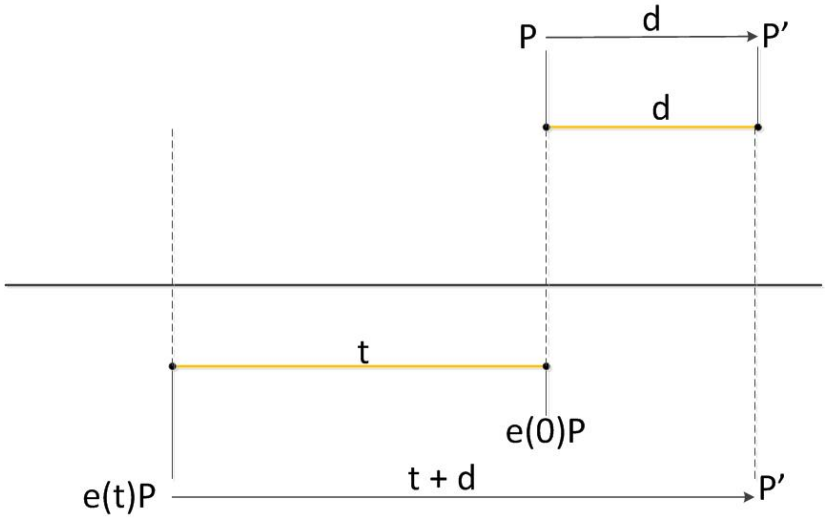


Figure 4.2: Depiction of the rule DEL-ADD which allows a delay prefixed process to delay beyond timeout.

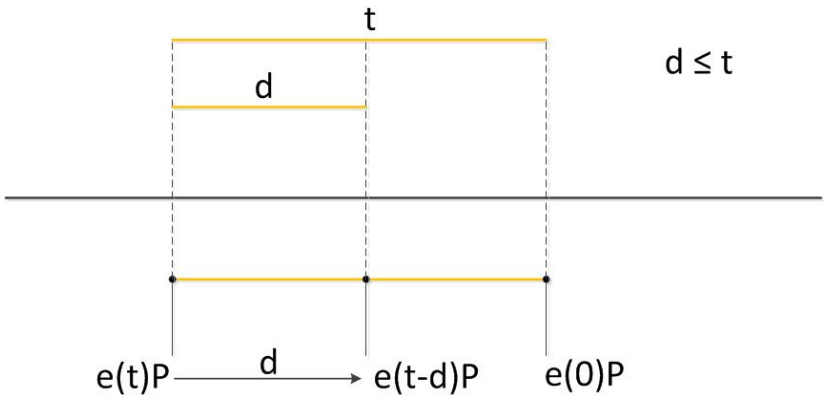


Figure 4.3: Depiction of the rule DEL-SUB, by which the time in a prefix decreases after a delay.

property of the language called maximal progress, which basically states that all processes ready to perform some internal computation should do so immediately. More about maximal progress later.

Table 4.5: Semantics of timed transitions in the software language, defined over processes, which are closed terms.

NIL	$\dfrac{}{\mathbf{0} \xrightarrow{d} \mathbf{0}}$
PFX-OUT-DEL	$\dfrac{}{c\langle\vec{e}\rangle!.\mathbf{P} \xrightarrow{d} c\langle\vec{e}\rangle!.\mathbf{P}}$
PFX-IN-DEL	$\dfrac{}{c\langle\vec{x}\rangle?.\mathbf{T} \xrightarrow{d} c\langle\vec{x}\rangle?.\mathbf{T}}$
THEN-T-DEL	$\dfrac{\mathbf{P} \xrightarrow{d} \mathbf{P}' \qquad [\![b_{exp}]\!] = \mathbf{true}}{\mathbf{if}\ b_{exp}\ \mathbf{then}\ \mathbf{P} \xrightarrow{d} \mathbf{P}'}$
THEN-F-DEL	$\dfrac{[\![b_{exp}]\!] = \mathbf{false}}{\mathbf{if}\ b_{exp}\ \mathbf{then}\ \mathbf{P} \xrightarrow{d} \mathbf{if}\ b_{exp}\ \mathbf{then}\ \mathbf{P}}$
CHO-DEL	$\dfrac{\mathbf{P}_1 \xrightarrow{d} \mathbf{P}_1' \qquad \mathbf{P}_2 \xrightarrow{d} \mathbf{P}_2'}{\mathbf{P}_1 + \mathbf{P}_2 \xrightarrow{d} \mathbf{P}_1' + \mathbf{P}_2'}$
PAR-DEL	$\dfrac{\mathit{noSyncSort}(\mathbf{P},\mathbf{Q},d) \qquad \mathbf{P} \xrightarrow{d} \mathbf{P}' \qquad \mathbf{Q} \xrightarrow{d} \mathbf{Q}'}{\mathbf{P} \mid \mathbf{Q} \xrightarrow{d} \mathbf{P}' \mid \mathbf{Q}'}$
APP-DEL	$\dfrac{\mathbf{N}(\vec{x}) \stackrel{\text{def}}{=} \mathbf{T} \qquad \mathbf{T}[\vec{e}/\vec{x}] \xrightarrow{d} \mathbf{P}'}{\mathbf{N}(\vec{e}) \xrightarrow{d} \mathbf{P}'}$
DEL-SUB	$\dfrac{d \leq t \qquad [\![e]\!] = t}{\varepsilon(e).\mathbf{P} \xrightarrow{d} \varepsilon(t-d).\mathbf{P}}$
DEL-ADD	$\dfrac{\mathbf{P} \xrightarrow{d} \mathbf{P}' \qquad [\![e]\!] = t}{\varepsilon(e).\mathbf{P} \xrightarrow{t+d} \mathbf{P}'}$
DEL-ILL	$\dfrac{[\![e]\!] = \bot}{\varepsilon(e).\mathbf{P} \xrightarrow{d} \varepsilon(e).\mathbf{P}}$

Let us now move on to the rules for the conditional term. The rule THEN-T-DEL, for the case where the guard is true, is straightforward: If the process **P** behind the guard can delay to some derivative, and the guard is true, then the entire conditional can also perform the same delay to the same derivative. For example, take the process

$$\mathbf{if}\ 1+7=8\ \mathbf{then}\ \varepsilon(30).\mathit{chan}!.\mathbf{0}$$

It is capable of performing the following transition:

$$\mathbf{if}\ 1+7=8\ \mathbf{then}\ \varepsilon(30).\mathit{chan}!.\mathbf{0} \xrightarrow{26} \varepsilon(4).\mathit{chan}!.\mathbf{0}$$

because the guard $1+7=8$ is assumed to evaluate to true in this example and because the inner process is capable of the same transition i.e.

$$\varepsilon(30).\mathit{chan}!.\mathbf{0} \xrightarrow{26} \varepsilon(4).\mathit{chan}!.\mathbf{0}$$

Note the term $chan!.\mathbf{0}$, which is a syntactic shortcut for $chan\langle\rangle!.\mathbf{0}$; we can output empty vectors across channels and when this is the case the brackets will be omitted after the channel name. If the guard in a conditional is false, then it behaves exactly like the nil process and allows an arbitrary amount of time to pass without evolving to anything new. This is behaviour captured by the rule THEN-F-DEL.

The rules CHO-DEL and PAR-DEL are similar. The premises of both rules require that all sub-processes can delay by the same amount of time d and the conclusion in both cases is that the top-level construct can delay by the same amount d. Notice that the passage of time preserves the choice construct and requires all of its sub-processes to delay together, whereas a discrete action collapses choice by following the evolution of just one sub-process. One reason for this is that it is desirable for the delay relation to be deterministic. This is a property known as time determinism, and is discussed later. Let us demonstrate the rule for parallel processes using the following process as an example:

$$chan1\langle 22\rangle!.\mathbf{0} \mid \varepsilon(34).chan1\langle x\rangle?.chan2\langle x\rangle!.\mathbf{0}$$

Now by PFX-OUT-DEL,

$$chan1\langle 22\rangle!.\mathbf{0}$$

can delay by any amount of time and remain unchanged. Also, by the subtraction law for delay prefixes,

$$\varepsilon(34).chan1\langle x\rangle?.chan2\langle x\rangle!.\mathbf{0}$$

can delay by any amount of time less than or equal to 34, and this time gets subtracted from the delay prefix in the derivative process; in particular, it can perform a delay of exactly 34 causing it to timeout. Hence the overall process can perform a delay of 34 and evolve as follows:

$$chan1\langle 22\rangle!.\mathbf{0} \mid \varepsilon(34).chan1\langle x\rangle?.chan2\langle x\rangle!.\mathbf{0} \xrightarrow{34} chan1\langle 22\rangle!.\mathbf{0} \mid \varepsilon(0).chan1\langle x\rangle?.chan2\langle x\rangle!.\mathbf{0}$$

Note that since the right hand process is now timed out, it is now input-enabled and thus a synchronisation can occur with a message passed from left to right:

$$chan1\langle 22\rangle!.\mathbf{0} \mid \varepsilon(0).chan1\langle x\rangle?.chan2\langle x\rangle!.\mathbf{0} \xrightarrow{\tau} \mathbf{0} \mid chan2\langle 22\rangle!.\mathbf{0}$$

For the choice rule, let us consider a different example, namely the process

$$\varepsilon(32).chan1!.\mathbf{0} + \varepsilon(14).chan2!.\mathbf{0}$$

Since both sub-processes can delay by 14 by the delay prefix subtraction rule, then so can the entire process. The resultant transition is

$$\varepsilon(32).chan1!.\mathbf{0} + \varepsilon(14).chan2!.\mathbf{0} \xrightarrow{14} \varepsilon(18).chan1!.\mathbf{0} + \varepsilon(0).chan2!.\mathbf{0}$$

Notice how the derivative is still a sum and also that it is now capable of an output on $chan2$ since the right hand prefix has timed out.

The rule DEL-ILL says that if an expression e is ill-defined, i.e. it does not evaluate to anything, then the process delay-prefixed with this expression can delay by any arbitrary amount and remain unchanged. Not in the premise the symbol $\perp$which simply means “undefined”. The reason for such a law is that it makes certain language properties easier to prove. This is a Coq technicality, and will not be discussed in detail here. It suffices to say that such a process can be considered as ”stuck” and so it should arguably have the same behaviours as the nil process.

Remark 4.8 (Trace Notation)**.** Often it is convenient to denote multiple sequential transitions using trace

notation. We write

$$\mathbf{P}_1 \xrightarrow{\alpha_1} \mathbf{P}_2 \xrightarrow{\alpha_2} \mathbf{P}_3 \ldots \xrightarrow{\alpha_n} \mathbf{P}_{n+1}$$

as shorthand for the individual transitions

$$\mathbf{P}_1 \xrightarrow{\alpha_1} \mathbf{P}_2$$

and

$$\mathbf{P}_2 \xrightarrow{\alpha_2} \mathbf{P}_3$$

and

$$\ldots$$

and

$$\mathbf{P}_n \xrightarrow{\alpha_n} \mathbf{P}_{n+1}$$

This notation applies not only to the current language of software processes but to any language whose semantics are in the form of labelled transitions over its terms. ▲

Example 4.9 (A Simple Timed Process)**.** Below is an example process $P(t)$, which is parametrised on a time variable t. The job of this process is to periodically read some integer value, bind it to the variable x and then output it every 10 time units. The parameter t records how long is left before the next read/output of the variable. This process also listens for external events *stop* and *start*. An input on the channel *stop* causes the process to enter a waiting state, which does nothing but listen on the channel *start*. An input on *start* causes the process to either leave the wait state or to zero its time parameter t, in either case the resultant process is one that is immediately willing to input and then output x.

Note that there is the implicit assumption of an environment here, with which external actions are postulated to occur; this may be another software process or something else entirely. Rather than dwell on a discussion of the environment here, we just assume it exists as some abstract entity capable of producing actions. The details of this environment are covered in Section 4.4, where software components are composed in the context of entities and networks.

We will now examine these behaviours by means of a trace. Let it be stressed that this is just one possible trace of the process that has been chosen to illustrate its main behaviours. Also, there is a slight abuse of notation in this example: we often replace a term which is the right hand side of a parametrised process definition by the process name, e.g. we would replace $stop?_\bullet.Q + \varepsilon(5)_\bullet.in\langle x\rangle?_\bullet.R(x) + Q$ with $P(5)$. This is strictly not correct but saves a lot of space and makes the example more readable. Similarly, we do not explicitly mention applications of the rules APP and APP-DEL even thought strictly speaking they are being applied for most transitions.

$$\begin{aligned} P(t) &\stackrel{\text{def}}{=} stop?_\bullet.Q + \varepsilon(t)_\bullet.in\langle x : \mathbb{N}\rangle?_\bullet.R(x) + Q \\ Q &\stackrel{\text{def}}{=} start?_\bullet.P(0) \\ R(x) &\stackrel{\text{def}}{=} out\langle x\rangle!_\bullet.P(10) \end{aligned}$$

We decide to start our trace with the process $P(8)$, which has 8 units of time remaining until the next read/output of the variable x. This process then delays by 5 time units. This is possible because in the definition of $P(t)$, the right hand side is a sum containing two input prefixed processes, which can delay arbitrarily by the rule DEL-IN and one delay prefixed process whose guard of 8 exceeds 5, and so can delay by the rule DEL-SUB. The overall sum of course can delay whenever its individual components can, as per the rule CHO-DEL. Hence we end up with the transition:

$$P(8) \xrightarrow{5} P(3)$$

Now suppose at this point, the environment signals on the channel *stop*. Then the first component of the sum

$stop?.Q$ reacts to this event and the entire process evolves to become Q:

$$P(3) \xrightarrow{stop?} Q$$

Note that the rule used here was CHO-DISC-L, in conjunction with PFX-IN-MSG. Now, in the stopped state Q, the process does nothing but idle and listen for a start event from the environment. Let us arbitrarily decide that Q decides to idle for 7 time units. Then by the rule PFX-IN-DEL, it simply delays and remains unchanged:

$$Q \xrightarrow{7} Q$$

Now suppose there is a start event signalled by the environment along the channel $start$. Then by the rule PFX-IN-MSG the process Q can evolve to become $P(0)$

$$Q \xrightarrow{start?} P(0)$$

The middle component of the sum constituting the process $P(0)$ is now timed out, rendering it capable of an input of some integer in place of the variable x. Let's choose the integer in question to be 9. Then we have the following transition:

$$P(0) \xrightarrow{in\langle 9\rangle?} R(9)$$

Notice in this case how the variable x has been replaced by the concrete value 9 after the input action has been performed. Now, the process $R(9)$ is ready to output the value 9, but it can still delay by an arbitrary amount, say 110, as per the rule PFX-OUT-DEL:

$$R(9) \xrightarrow{110} R(9)$$

Finally, we assume that the process performs its output via the following transition:

$$R(9) \xrightarrow{out\langle 9\rangle!} P(10)$$

Note that there was a subtlety in the behaviour of this process that may not have been clear at first sight. This was in the fact that the process could delay arbitrarily between input and output actions, meaning that the intended period of 10 time units was not always achieved. However, by the predicate *noSync*, this process, if placed in parallel composition with another process listening on the channel *out*, would no longer be capable of performing this delay and would be force to output the value bound to x immediately after it is input.

This example appears again in a slightly more complex form in the exposition of the protocol, Section 5.3.

▲

4.1.6 Sanity Checks for the relation *Sort*

Recall the relation *Sort* from Definition 4.6. The idea behind this relation is to approximate the discrete half actions, i.e. inputs and ouptuts, available to a process within a given time. In some of the following theorems the restriction of discrete actions to just input and output, i.e. the exclusion of τ actions, is given by $\delta \neq \tau$. It is used in the delay semantics in order to ensure that whenever complementary actions are available for concurrent processes, then time cannot pass. However, no verification of *Sort* has been given so far to show that it is indeed a "sane" choice of relation. In this section, a number of theorems will be stated, showing that *Sort* does indeed behave in tune with our expectations. The theorems have all been proved in the file "SoftwareLanguage.v" of the Coq model. The proofs are quite uninteresting, consisting mostly of either structural induction on *Sort* or rule induction on the delay relation for software processes, hence they are omitted here. The reader interested in the details of these proofs is referred to the Coq model.

Theorem 4.1 (Enabled implies Sort)**.** *If a process is enabled on a half action δ, then for all delays d, δ is*

related to that process by Sort with time d.

$$\mathbf{P} \xrightarrow{\delta} \Rightarrow \delta \neq \tau \Rightarrow \delta \ \mathit{Sort}_d \ \mathbf{P}$$

Theorem 4.2 (Sort Derivative implies Source)**.** *If δ is related by Sort to $\mathbf{P}''$ with delay d' and $\mathbf{P}$ transitions to $\mathbf{P}'$ by d, then δ is related to $\mathbf{P}$ with delay $d + d'$.*

$$\delta \ \mathit{Sort}_{d'} \ \mathbf{P}' \Rightarrow \mathbf{P} \xrightarrow{d} \mathbf{P}' \Rightarrow \delta \ \mathit{Sort}_{d+d'} \ \mathbf{P}$$

Theorem 4.3 (Sort Less than or Equal)**.** *If δ Sort_d $\mathbf{P}$ and $d \leq d'$, then δ $\mathit{Sort}_{d'}$ $\mathbf{P}$. In other words, Sort is preserved by extending the delay parameter. This is in keeping with our intuitions in that Sort captures all the actions that can be performed within some delay d: clearly extending the delay does not exclude any more actions.*

$$\delta \ \mathit{Sort}_d \ \mathbf{P} \Rightarrow d \leq d' \Rightarrow \delta \ \mathit{Sort}_{d'} \ \mathbf{P}$$

Theorem 4.4 (Sort Time Out)**.** *If a process $\mathbf{P}$ is guarded by a value that is timed out, then all actions related to $\mathbf{P}$ by Sort are also related to the guarded process $\varepsilon(e).\mathbf{P}$.*

$$[\![e]\!] = 0 \Rightarrow \delta \ \mathit{Sort}_d \ \mathbf{P} \Rightarrow \delta \ \mathit{Sort}_d \ (\varepsilon(e).\mathbf{P})$$

Theorem 4.5 (Sort Complete)**.** *Recall the initial intuition behind developing the Sort relation. This is that it should approximate the discrete half actions available to a process within the scope of a certain delay. This result is stated here and formally proved in Coq, using the preceding results. It says that if a half action can be done now by a process $\mathbf{P}$ or at some derivative state in the future d' time units away less than d, then that half action is related to the process via Sort. The converse of this theorem however is not true: Sort over-approximates. Too see why this is the case, refer to Example 4.10.*

$$\delta \neq \tau \wedge (\mathbf{P} \xrightarrow{\delta} \vee \exists\, d', \mathbf{P}', d' < d \wedge \mathbf{P} \xrightarrow{d'} \mathbf{P}' \wedge \mathbf{P}' \xrightarrow{\delta}) \Rightarrow \delta \ \mathit{Sort}_d \ \mathbf{P}$$

Example 4.10 (Over-approximation of *Sort*)**.** In this example, a process is shown to which an action is related via the *Sort* relation, but which cannot do that action at any future state. This renders *Sort* an unsound i.e. over-approximating relation with respect to the converse of Theorem 4.5. The example process is a sum of two processes but a parallel construct would have been equally suitable for the purposes of this demonstration. Below is the process

$$\varepsilon(2).d!.\mathbf{0} + (c!.\mathbf{0} \mid c?.\mathbf{0})$$

The channel d is available for output by the LHS of this sum after a delay of 2 time units, and an output on d is related to the LHS by the sort relation with a delay parameter of any amount greater than 2. For example, we could take the time parameter to be 3, giving

$$d! \ \mathit{Sort}_3 \ \varepsilon(2).d!.\mathbf{0}$$

Now, the inductive rules for sum processes in the definition of *Sort* dictate that whenever the relation holds for either component of a sum, then it holds for the whole sum. Since it is the case that sort holds for the LHS, it can be shown to hold for the entire sum i.e.

$$d! \ \mathit{Sort}_3 \ \varepsilon(2).d!.\mathbf{0} + (c!.\mathbf{0} \mid c?.\mathbf{0})$$

However, there is no future state of this process that can be reached by a delay of less than 3 such that the action $d!$ is available to perform. In fact, this process is not capable of delay at all. This is because there is a synchronisation on the channel c immediately available in the right hand side process (RHS); hence *noSyncSort* does not hold and the RHS cannot delay, which in turn means the entire sum can't delay because this would

require a delay from both components.

The irony of this example is that it is the very inclusion of *Sort* in the rule PAR-DEL that renders the RHS incapable of delay: *Sort* contributes to its own unsoundness. This is what complicates formulating correctness properties for *Sort*. Such properties will inevitably contain some reference to the delay semantics, which are in turn reliant on the definition of *Sort*, so there is an inescapable circularity involved.

At a first glance, this over-approximating behaviour of *Sort* seems problematic. *Sort* is used to inhibit certain delays from occurring if a synchronisation is available. However, since some actions predicted by *Sort* are not actually reachable within a future state of the process, the question is, do "valid" delays get inhibited? Of course, to answer this, the very notion of what a "valid" delay is would need to be explored. Intuitively though, the answer to this question is no, this example is not as problematic as it seems.

By looking again at the example, it can be seen that the only reason the action in question isn't available in a future state is because there is *already* a synchronisation available. Now, it seems a valid claim that there will always be some *first* such synchronisation possible for a given process, given that there was indeed *some* synchronisation at all at a future delay derivative state. That is, either synchronisation is possible now or there is some delay that can transpire after which a synchronisation becomes available, but before which synchronisation cannot occur. Before this first synchronisation, it seems the relation *Sort* behaves in the expected way, i.e. the converse of Theorem 4.4 should hold- a partial "soundness" result. However, to formalise this would be quite difficult. Instead, we gain confidence in the correctness of *Sort* through the emergent properties given in Section 4.1.7, particularly the properties of patience and maximal progress given in Theorem 4.7 and Theorem 4.6 respectively. Together, these imply that a process is never "stuck" i.e. it is always capable of either delaying or doing a τ action. ▲

4.1.7 Software Language Properties

Presented here are fully machine checked results demonstrating properties of the software language. These results increase our confidence in the overall "sanity" of the language, in the sense that they demonstrate the conformance of the language to many of our expectations. In other words, they show that the language does indeed behave as it "should" behave according to a number of our intuitions. Each result is stated as a theorem, with some accompanying explanation. The proofs of these properties are omitted, but can be found in the file "SoftwareLanguage.v" of the Coq model. Also proved in that file are less significant results that may still be of some interest.

Theorem 4.6 (Maximal Progress)**.** *Whenever a process* $\mathbf{P}$ *can do a silent action, it cannot delay.*

$$\mathbf{P} \xrightarrow{\tau}\Rightarrow \neg\mathbf{P} \xrightarrow{d}$$

Theorem 4.7 (Patience)**.** *If a process cannot perform a* τ *action, then it can delay by some amount* d*.*

$$\neg\mathbf{P} \xrightarrow{\tau}\Rightarrow \exists d, \mathbf{P} \xrightarrow{d}$$

Theorem 4.8 (Time Split)**.** *If a process* $\mathbf{P}$ *can delay by* d'' *to become* $\mathbf{P}''$*, and there is some* d *and* d' *such that* $d'' = d + d'$*, then there is some* $\mathbf{P}'$ *such that* $\mathbf{P}$ *can delay by* d *to get to* $\mathbf{P}'$*, and* $\mathbf{P}'$ *can delay by* d' *to* $\mathbf{P}''$*.*

$$\mathbf{P} \xrightarrow{d''} \mathbf{P}'' \Rightarrow d'' = d + d' \Rightarrow \exists \mathbf{P}', \mathbf{P} \xrightarrow{d} \mathbf{P}' \wedge \mathbf{P}' \xrightarrow{d'} \mathbf{P}''$$

Theorem 4.9 (persistence)**.** *If a process can delay to another process, then all discrete actions available to the original process are available to the derivative. In other words, the possibility of performing a discrete action is preserved by the delay transition relation.*

$$\mathbf{P} \xrightarrow{d} \mathbf{P}' \Rightarrow \mathbf{P} \xrightarrow{\delta}\Rightarrow \mathbf{P}' \xrightarrow{\delta}$$

Theorem 4.10 (Delay Additivity)**.** *If* $\mathbf{P}$ *can delay to* $\mathbf{P}'$*, and* $\mathbf{P}'$ *can in turn delay to* $\mathbf{P}''$*, then* $\mathbf{P}$ *can delay to* $\mathbf{P}''$ *by the sum of the two original delays.*

$$\mathbf{P} \xrightarrow{d} \mathbf{P}' \Rightarrow \mathbf{P}' \xrightarrow{d'} \mathbf{P}'' \Rightarrow \mathbf{P} \xrightarrow{d+d'} \mathbf{P}''$$

Theorem 4.11 (Time Determinacy)**.** *The passage of time in this universe is linear, i.e. any process can delay by an amount d to at most one derivative process. The formal statement of this says that whenever two delay transitions can be shown from the same source process with the same delay value, then the derivatives of the transitions are equal.*

$$\mathbf{P} \xrightarrow{d} \mathbf{P}'_1 \Rightarrow \mathbf{P} \xrightarrow{d} \mathbf{P}'_2 \Rightarrow \mathbf{P}'_1 = \mathbf{P}'_2$$

4.1.8 Zeno Behaviours of Software Processes

In relation to formalisms dealing with real time, "zeno" behaviour is generally considered problematic. In [Henzinger, 2000], non-zenoness is defined as: "A hybrid automaton is nonzeno if it cannot prevent time from diverging". While this definition applies to hybrid automata, it can be easily generalised to any real-time formalism. Clearly, nonzenoness is favourable in a formalism that claims to model the real world: the idea that some system can affect the passage of time seems absurd.

The double negative in the above definition of non-zenoness is somewhat confusing. A simpler approach would be to define zenoness in terms of some entity that *can* prevent time from diverging. However, the notion of time diverging is still ambiguous. In terms of the language presented here, the concept of divergence/convergence seems only to make sense in relation to infinite traces i.e. infinite sequences of states with consecutive states linked by actions in the LTS. Zenoness would then amount to the convergence of the sum of all the delays over such a trace. This is defined more formally in Definition 4.11.

It is indeed the case that this language admits zeno behaviours; this is shown in Example 4.12. It may now seem appropriate to ask, is this a problem? The answer to this is not clear until further discussions are given. In Remark 4.21, zenoness is discussed at the entity language level. Then the overall implications of zeno behaviours are discussed in Section 8.2.4.

Definition 4.11 (Zeno Software Trace)**.** Zenoness is defined in terms of infinite traces; it would not make sense to talk about time diverging on finite traces since a finite amount of delays implies a finite amount of time has passed. Recall from Remark 4.8 that a trace is a sequence of processes interleaved with actions. We extend this to infinite traces by assuming these sequences continue indefinitely i.e.

$$\mathbf{P}_1 \xrightarrow{\alpha_1} \mathbf{P}_2 \xrightarrow{\alpha_2} \mathbf{P}_3 \ldots \xrightarrow{\alpha_n} \mathbf{P}_{n+1} \xrightarrow{\alpha_{n+1}} \ldots$$

Now, it is possible to talk about the divergence or convergence of time in such a trace. The time that has passed at any point in the trace is the sum of delays up to that point. If this total time is bounded from above by a fixed constant for all indices in the trace, then time converges. Another way of looking at this is that the possibly infinite sum of all the delays in the infinite trace converges.

Equivalently, rather than extract the delays, we could define a function *dur* giving a time value to every action and then add up the times of all the actions. The function is the identity function over delays and maps all discrete actions to 0 i.e. $dur(d) = d$ and $dur(\delta) = 0$. Then the infinite sum of interest would be

$$dur(\alpha_1) + dur(\alpha_2) + \ldots + dur(\alpha_n) + \ldots$$

A convergent sum is then deemed zeno. There are two types of zenoness to consider here. The first is that there are only a finite number of delays in the trace. Then clearly the above sum, which is equivalent to the sum of all the delays, converges. In this case, there will be some infinite suffix to the trace after a finite prefix consisting of only discrete actions. A special sub-case is that there are *no* delays in the original trace. Regardless, there will always be an infinite suffix of only discrete actions, which is possibly the trace itself.

The second type of of zenoness involves an infinite amount of delays whose sum converges. For example, the delays could constitute a geometric sequence with a ratio of less than 1.

■

Example 4.12 (Zeno Software Traces)**.** Now, it may be argued that the definition of zeno traces given in Definition 4.11 is too lax: zeno traces may consist of only half actions. For example, a process IN may be polling for input on a channel as defined below

$$IN \stackrel{\text{def}}{=} c?_\bullet IN$$

Clearly, this process is capable of the infinite trace

$$IN \xrightarrow{c?} IN \xrightarrow{c?} IN \ldots \xrightarrow{c?} IN \ldots$$

However, it is arguable that such a trace is not so absurd. The intuition is that input actions alone do not have real meaning unless there is some corresponding output action. Thus the definition of zenoness can be tightened to include only τ actions and delays in its traces.

Now, it is still possible to construct a reasonably simple zeno software process, even with this restriction to τ actions. To do so, simply take the above process IN and put in parallel with some other process OUT whose job it is to infinitely output on the same channel upon which IN is listening.

$$OUT \stackrel{\text{def}}{=} c!_\bullet OUT$$

The zeno process is then the parallel construct $\mathbf{N} \mid OUT$. We refer to it here as $ZENO$ for shorthand. By virtue of one process being capable of input on the channel c and the other being capable of output on the same channel, this process can always do a τ action. Thus it can do an infinite sequence of these τ actions, and no time at all passes.

$$ZENO \xrightarrow{\tau} ZENO \xrightarrow{\tau} ZENO \ldots \xrightarrow{\tau} ZENO \ldots$$

The question that arises now is whether this possibility of zenoness in the language is purely a theoretical artefact or if it has any real consequences in the later development of the Comhordú protocol? To answer this question, further exploration of zenoness is necessary, culminating in Section 8.2.4. ▲

4.2 Interface Language

The language presented here simulates an interface to the Space-Elastic Model (SEM), an underlying model assumed to be present in any Comhordú system. The implementation details of SEM are not modelled here as they are too low level. Rather, what is assumed is a set of guarantees provided to higher level software such as the Comhordú protocol. Let us now discuss what these guarantees are before showing how they are realised.

- Bounded message delivery to an elastic region: After a predetermined time bound $msgLatency$, messages sent by an entity will be delivered to all neighbouring entities within some region of space surrounding it. This region of space is called the coverage. At the cost of keeping the time bound for message delivery fixed, coverage may shrink or stretch over time, hence the term “space-elastic”. Since coverage variation is subject to unknown environmental factors, which are essentially random, we assume randomly fluctuating coverage in this model.

- Bounded notification of message coverage to the sender: The chief guarantee of SEM is that a sender is notified of the coverages to which its messages were sent. This notification occurs within a bounded time $adaptNotif$ of the delivery of the corresponding message.

Remark 4.13 (Modelling SEM: Design Choices)**.** Once the behaviours of SEM have been identified, the question that arises is "how is SEM to be modelled?" A number of solutions come to mind here. Let us briefly mention them here and then discuss them in slightly more detail:

1. Encode the SEM behaviours directly into the software language. A driving factor for this choice is minimising the number of languages involved in the model. However, the downside is that the software component needed to model the required behaviours would be unnatural- a "hack" so to speak.

2. Give a list of accompanying axioms to the language stating certain behaviours that must be obeyed e.g. "If message x was sent at time t then all entities will receive it at time t'".

3. Extend the language: invent another sub-language with operational semantics embodying the required behaviours.

To see why Item 1 would not work, consider incoming messages from the environment. There is no bound of the number of these messages that could be in transit at any given time. Since this component aims to model messages in transit, it would need to buffer such messages and then forward them after the message latency $msgLatency$ elapsed. However, there is no conceivable way of doing this without giving the software process unbounded spawning behaviour, with a new process being spawned each time a message is sent to model that message in transit. This would give rise to a much more complex LTS for the software component, not to mention the confusion arising from the entanglement of conceptually separate components.

The purely axiomatic approach is attractive: it abstracts from any implementation details and gives properties that are directly usable for the higher level proofs. In retrospect, it seems that axioms may have been the best choice; a number of supporting results simply serve as buffers between the low-level operational specification and the high level proofs that may well have simply been assumed as axioms. However, in the nebulous stages of design when these choices were being considered, it seemed that a departure from the structural operational approach would cause problems. The intuition was that the combination of axioms and structural operational semantics would introduce a certain heterogeneity to the language that would make it more complicated to work with.

The approach that is taken is to extend the language. It is arguable that this is a compromise between the other two solutions: the structural operational approach is maintained but a sub-language is defined which is tailored to the specific requirements of SEM, in contrast to attempting their realisation in the unsuitable software language.

Of course, the options discussed here are not exhaustive, but merely constitute the ideas that led to the current design. It is certainly conceivable that there are "better" designs possible. ▲

It is now time to present the interface language that has been chosen to model SEM. Looking at Figure 4.1 it can be seen that the interface component of an entity can communicate with both the software component and with the outside world. As such, it acts in part as a buffer relaying messages from the environment to the software controller of the entity. The interface also provides notifications of message coverage to the software process. An interface $\mathbf{I}$ is modelled as a triple of multisets, which are sets whose elements are given a multiplicity. For simplicity we use lists instead of multisets, though let it be stressed that the order of elements in the lists doesn't matter. The three components of the interface are as follows:

- $\mathbf{L_i}$ A list of buffered incoming messages which have just been received. These messages are ready for consumption by the software process. However, if they are not consumed immediately, they will be dropped: this models the possibility of message loss.

- $\mathbf{L_o}$ A list of timestamped outgoing messages ready for broadcast to the environment. Each message in the list is paired with a timestamp indicating the time pending before its delivery.

- $\mathbf{L_n}$ A list of timestamped notification messages pending delivery back to the software process. These messages notify the software process of the coverages of its outgoing messages.

Remark 4.14 (Interfaces for each individual entity?)**.** One thing that may seem odd about the interface language is that each entity is given its *own* interface component. Conceptually, this does not seem right: messages in transit surely belong to a "common space" rather than a buffer within the sender. Indeed, but for practical reasons, the interface-per-entity approach has been adopted. For example, it ensures that messages are never returned to senders; this would add further complications to the system e.g. entities having to add their identities to messages and then ignore messages whose identity matched their own. ▲

Figure 4.4 is schematic diagram of the interface component, showing the various lists used to buffer messages. Notice the directions of the various arrows connecting the lists to the distinguished channels, conveying the direction of message flow.

- $\mathbf{L_i}$ receives messages from the environment over the channel *chanIOE*. Messages are buffered in this input list and then forwarded on to the channel *chanInP*, to which the software process may listen. If the software process is not willing to consume a message straight away, it is dropped from the input list. This allows for message loss, at the application layer rather than the physical layer.

- $\mathbf{L_o}$ behaves in a very similar way to $\mathbf{L_i}$. Messages this time are received from the software process over *chanOutP* and then buffered. On receipt, a message is timestamped by the constant *msgLatency*, held in the buffer for that amount of time, and then forwarded out to the environment over the channel *chanIOE* once the timestamp has elapsed. Simultaneously as the message is output to the environment, it gets appended to the notification list, augmented with information about the coverage to which the message was sent.

- $\mathbf{L_n}$ When messages are appended to the notification list, they are timestamped by the constant $adaptNotif$, i.e. the notification lag time. Once a message times out, it is forwarded along the channel *chanAN* to the software process, which can in turn use the extra information regarding the coverage of the message.

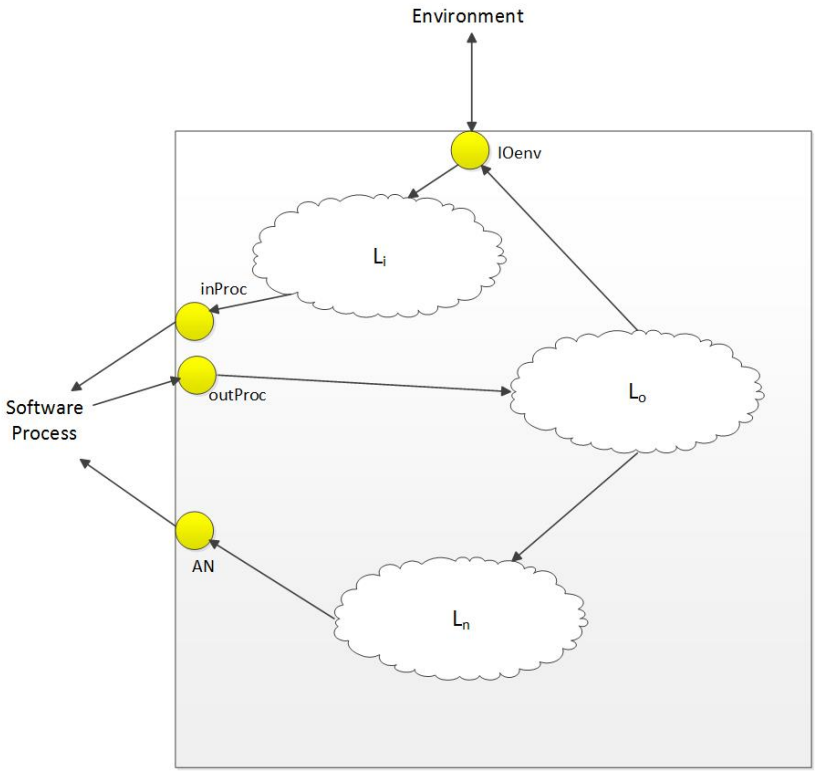


Figure 4.4: The interface component of an entity, with the input, output and notification lists shown.

Now, since the input, output and notification multisets are so similar in their behaviour, it makes sense to give general semantics for multisets of timestamped elements and modify these semantics accordingly to obtain the individual semantics for each specific multiset. Again, we are modelling multisets as lists, even though semantically the order of elements in the list doesn't matter, only their multiplicity does. In what follows, we will use the metavariables L and L' to denote arbitrary lists of timestamped elements, which we will call timed lists. We will give a general semantics that will be shared by all instances of these lists. The approach is much like that of abstraction and inheritance in object oriented programming: define a superclass with some general behaviour and then extend this in various directions by adding new specific behaviours for the subclasses. Let us now present a syntax and semantics for these lists.

Table 4.6: Syntax for lists of timestamped elements.

$$L ::= [] \mid \langle \vec{v} \rangle_t \frown L$$

4.2.1 Timed Lists

Table 4.6 presents the syntax of timed lists, which is fairly straightforward. A timed list is either the empty list, or it is a timestamped element concatenated onto the beginning of another timed list. Notice the type of the elements of the list. They consist of a vector of values $\vec{v}$, all of the base type, augmented with a timestamp t. Informally, the idea is that all elements in a list will delay simultaneously with their timestamps decreasing at the same rate until some timestamp reaches zero. At that point, the element in question is ready to be output. We would also like these lists to be capable of inputting vectors of values $\vec{v}$, stamping them with some initial predetermined time bound T to create an element, and then adding this element to the list. For these general semantics, the time bound T will be left as a parameter that will be instantiated with different values to obtain the semantic variations for input, output and notification lists. Let us now formalise these notions that have been discussed.

Table 4.7: Semantics for timed lists.

BUFF	$\dfrac{}{L \xrightarrow{\langle \vec{v} \rangle ?} \langle \vec{v} \rangle_T \frown L}$
FWD-HD	$\dfrac{}{\langle \vec{v} \rangle_0 \frown L \xrightarrow{\langle \vec{v} \rangle !} L}$
FWD	$\dfrac{L \xrightarrow{\langle \vec{v}_1 \rangle !} L'}{\langle \vec{v}_2 \rangle_t \frown L \xrightarrow{\langle \vec{v}_1 \rangle !} \langle \vec{v}_2 \rangle_t \frown L'}$
DEL-EMP	$\dfrac{}{[] \xrightarrow{d} []}$
DEL	$\dfrac{L \xrightarrow{d} L' \qquad d \leq t}{\langle \vec{v} \rangle_t \frown L \xrightarrow{d} \langle \vec{v} \rangle_{t-d} \frown L'}$

Table 4.7 presents the semantics for general timed lists. Such a list receives messages, buffers them for a certain time and then forwards them on to some destination. Messages are added to the buffer with an initial timestamp of T. The buffer may continue inputting new messages at any time and may delay as long as all of its elements still have some positive timestamp remaining. Once the timestamp of an element reaches zero, this element must be output i.e. time can no longer pass when there are some elements in the list which have timed

out.

The rule BUFF is the only rule that mentions the time bound T upon which these semantics are parametrised. It says that a list is always capable of inputting some vector of values $\vec{v}$ and appending the timestamped element $\langle \vec{v} \rangle_T$ to front of the list. Of course, since we are only using the list as a multiset, the order of the elements does not matter, and we could have just as easily have chosen to add a new element anywhere in the list. However, clearly adding to the front of the list is the most convenient practical approach.

The rule FWD-HD says that when the head of the list has timed out, it may be output. The rule FWD says that whenever one list can output a vector of values, then so can the list obtained by concatenating any other vector of values on to the original list. Together, these two rules imply that *any* element in the list whose timestamp has reached zero may be output, demonstrating at least in part that element order is of no relevance here. Note that the existence multiple timed out elements in a list gives rise to non-determinacy: any one can be output.

The rule DEL-EMP allows the empty list to delay by any arbitrary amount of time. The rule DEL allows a delay to happen to a non-empty list $\langle \vec{v} \rangle_t \frown L$ if firstly that delay can happen to its tail L and secondly if the delay is less than the timestamp of the head of the list. The derivative list is obtained by taking the derivative of the tail L' and appending a modified head where the timestamp has been decremented by exactly the delay that took place i.e. the new head becomes $\langle \vec{v} \rangle_{t-d}$. Together with DEL-EMP , this rule implies that when a delay happens, all elements in the list simultaneously decrement their timestamps by that delay.

Recall that the interface is modelled as a triple of lists $\mathbf{L_i}$, $\mathbf{L_o}$ and $\mathbf{L_n}$. Syntactically, these list are the very same as timed lists i.e. they are lists of timestamped vectors of values. However, their individual semantics each varies slightly from the general semantics of timed lists. Table 4.8 presents the syntax for an interface $\mathbf{I}$. The semantics of the interface will be given in terms of the semantics of the three sub-components. These in turn will inherit the general semantics of timed lists that have just been presented, with some modifications for specific behaviours and an instantiation of the parameter T.

Table 4.8: Syntax for the interface language.

$\mathbf{I} ::= (\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n})$

For input lists, we take $T = 0$. In other words, on entry to the input list, elements are stamped with a time of 0. Note that this renders the delay rule DEL redundant: all values in the list have a timestamp of zero while this rule requires the delay, a strictly positive number, to be less than or equal to each timestamp, therefore the rule will never be applicable to an input list. Instead, another rule is introduced to capture the delay behaviour of input lists, the rule DEL-IN of Table 4.9. This rule says that an input list may delay at any time and drop any elements it may have buffered. The intuition behind this rule is that, if the software process is not listening to incoming messages at a particular time, and time is allowed to elapse, then these messages are simply lost. It may be of interest to note that this rule in fact subsumes the rule DEL-EMP ; hence the only rules from the original timed list semantics that apply to input lists are BUFF, FWD-HD and FWD.

Table 4.9: Additional rule for the input list semantics.

DEL-IN	$\dfrac{}{\mathbf{L_i} \xrightarrow{d} []}$

The semantics for the output list $\mathbf{L_o}$ are exactly the semantics of general timed lists with the time bound instantiated as $T = msgLatency$. The constant $msgLatency$ is an upper bound on the time it takes to deliver a message. Recall that this time bound is fixed at the expense of an elastic region of delivery called the coverage.

For the notification list $\mathbf{L_n}$, the time bound parameter is set as $T = adaptNotif$. The constant $adaptNotif$

is an upper bound on the time it takes for an entity to be notified of the coverage of a message after that message has been delivered. The two additional rules for notification lists are both delay rules. The first, DEL-NOTIF-DROP, says that if a message in the list times out, i.e. its timestamp reaches zero, then it can simply be dropped from the list. This is in contrast to the output list which will not allow time to pass until all its timed out messages are sent. The behaviour is more like that of the input list: again, we assume that if the software process is not listening for a message then that message is simply lost. The rule DEL-NOTIF-ADD allows delays to be added together. Without this rule, delays would be chopped up every time an element in the list timed out. Such a rule is unnecessary for the output list, which must halt at every element timeout.

Table 4.10: Additional semantics for the notification list.

DEL-NOTIF-DROP	$\dfrac{\mathbf{L_n} \xrightarrow{d} \mathbf{L_n}'}{\langle \vec{v} \rangle_0 \frown \mathbf{L_n} \xrightarrow{d} \mathbf{L_n}'}$
DEL-NOTIF-ADD	$\dfrac{\mathbf{L_n} \xrightarrow{d} \mathbf{L_n}' \qquad \mathbf{L_n}' \xrightarrow{d'} \mathbf{L_n}''}{\mathbf{L_n} \xrightarrow{d+d'} \mathbf{L_n}''}$

We will now combine these semantics of input, output and notification lists to obtain an overall semantics for interfaces. After that, we will return to timed lists with some examples of their behaviour in Section 4.2.3.

4.2.2 Interface Semantics

We are now ready to join the individual semantics for the input, output and notification lists to yield an overall semantics for the interface component of our language. The rules in Table 4.11 mostly lift inputs, outputs and delays from the sub-component lists to the interface component, which is a triple of lists. This lifting usually involves piping inputs/outputs through the correct channels, as in Figure 4.4. Let us now discuss the rules in the semantics for the interface.

The rule DEL says that if all three lists can delay by some amount d, then so can the entire interface. The rules IBUFF-IN and IBUFF-OUT deal with the case where a message is added to a list. For IBUFF-IN, the message comes from the environment over the channel *chanIOE* and gets added to the input list $\mathbf{L_i}$ which then evolves to $\mathbf{L_i}'$. For IBUFF-OUT, the message comes from the software component over the channel *chanOutP* and gets added to the list $\mathbf{L_o}$ with timestamp *msgLatency*. The rules IFWD-IN and IFWD-OUT are similar in that they deal with the output of messages from lists. IFWD-IN is straightforward: whenever the input list is willing to send out a message, this message can be sent to the software process over the channel *chanInP*. IFWD-OUT is slightly more involved. The premises assert that $\mathbf{L_o}$ is able to output a value, and that $\mathbf{L_n}$ is able to input that vector of values prefixed with some measure of coverage r. The conclusion says that the vector of values with the coverage appended can be output over the channel *chanIOE*. Note that the value r is not bound by anything in this law; *any* value for r will suffice. The choice of a value for r introduces non-determinism into this rule, modelling the fact that message coverage is essentially random. The conclusion of this law also implies that the message, with the coverage appended, is added to the notification list. Once a delay of $adaptNotif$ elapses, this message is then sent back to the software process so that it may analyse the coverage to which the message was sent, which is exactly what the rule IFWD-NOTIF says.

Remark 4.15 (Flattening Coverage Information)**.** When a message is sent out to a certain range r, the same message, prefixed with r is added to the notification list for forwarding back to the software process in due time. Now, the range r is coverage information, metadata about the state of delivery of the message. Thus, it ideally should be kept separate from the message rather than simply adding it to the front of the list constituting the message. However, this would require a new structure to be defined such as a coverage-stamped message like $\langle \vec{v} \rangle_r$, and this in turn would require the use of a new type of list other than a simple timed list. For practical purposes this is avoided, and the coverage information is added to the beginning of the message. The

responsibility of extracting this information is thus left with the software process. ▲

Table 4.11: Semantics for the interface.

DEL	$\dfrac{\mathbf{L_i} \xrightarrow{d} \mathbf{L_i}' \qquad \mathbf{L_o} \xrightarrow{d} \mathbf{L_o}' \qquad \mathbf{L_n} \xrightarrow{d} \mathbf{L_n}'}{(\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}) \xrightarrow{d} (\mathbf{L_i}', \mathbf{L_o}', \mathbf{L_n}')}$
IBUFF-IN	$\dfrac{\mathbf{L_i} \xrightarrow{\langle \vec{v} \rangle ?} \mathbf{L_i}'}{(\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}) \xrightarrow{chanIOE\langle \vec{v} \rangle ?} (\mathbf{L_i}', \mathbf{L_o}, \mathbf{L_n})}$
IFWD-IN	$\dfrac{\mathbf{L_i} \xrightarrow{\langle \vec{v} \rangle !} \mathbf{L_i}'}{(\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}) \xrightarrow{chanInP\langle \vec{v} \rangle !} (\mathbf{L_i}', \mathbf{L_o}, \mathbf{L_n})}$
IBUFF-OUT	$\dfrac{\mathbf{L_o} \xrightarrow{\langle \vec{v} \rangle ?} \mathbf{L_o}'}{(\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}) \xrightarrow{chanOutP\langle \vec{v} \rangle ?} (\mathbf{L_i}, \mathbf{L_o}', \mathbf{L_n})}$
IFWD-OUT	$\dfrac{\mathbf{L_o} \xrightarrow{\langle \vec{v} \rangle !} \mathbf{L_o}' \qquad \mathbf{L_n} \xrightarrow{\langle r, \vec{v} \rangle ?} \mathbf{L_n}'}{(\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}) \xrightarrow{chanIOE\langle \vec{v} \rangle r !} (\mathbf{L_i}, \mathbf{L_o}', \mathbf{L_n}')}$
IFWD-NOTIF	$\dfrac{\mathbf{L_n} \xrightarrow{\langle \vec{v} \rangle !} \mathbf{L_n}'}{(\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}) \xrightarrow{chanAN\langle \vec{v} \rangle !} (\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}')}$

4.2.3 Timed List Examples

We now return to timed lists in order to demonstrate with the aid of simple examples the differences in behaviour between the input, output and notification lists. This section has been intentionally placed after the basic presentation of the language as the examples herein are quite basic. The reader who is reasonably satisfied in their understanding of the interface language may skip this section.

We begin by noting that the output list behaves exactly as a general timed list with the parameter T set to *msgLatency*. Bearing this in mind, we will give an example demonstrating the behaviour of a general timed list from which that of an output list will follow. To maintain generality for this example, we do not instantiate the parameter T to any particular value. Similarly, we do not specify the exact values of the elements being input/output from any of the lists since this level of detail is irrelevant. Nor do we specify the exact value of the delay d that occurs; rather we just assume that $d < T$, satisfying one premise of the rule DEL[2].

Example 4.16 (General Timed List Example)**.** Figure 4.5 depicts an example of the behaviour of a general timed list. In this example and the others to follow in this section, we will examine a sequence of possible transitions. This is often called a run, a computation, an execution or a trace. It is important to note that in general, a run is just one possible sequence of actions that can be performed. The idea here is to choose a sequence that demonstrates all the features of the component in question. We start with the empty list $[]$ and allow it to input some vectors $\vec{v}_1$ and $\vec{v}_2$. By two sequential applications of the rule IBUFF-IN we have

$$[] \xrightarrow{\langle \vec{v}_1 \rangle ?} \langle \vec{v}_1 \rangle_T \frown [] \xrightarrow{\langle \vec{v}_2 \rangle ?} \langle \vec{v}_2 \rangle_T \frown \langle \vec{v}_1 \rangle_T \frown []$$

You may notice here the use of trace notation, which is explained in Remark 4.8. Now our timed list contains two elements both timestamped with the initial bound T. All it can do in this state is allow time to pass until one or more of its elements times out. A combination of the rules DEL-EMP and DEL can be applied to yield the transition

$$\langle \vec{v}_2 \rangle_T \frown \langle \vec{v}_1 \rangle_T \frown [] \xrightarrow{d} \langle \vec{v}_2 \rangle_{T-d} \frown \langle \vec{v}_1 \rangle_{T-d} \frown []$$

[2] In fact, the premise demands that the delay be less than or equal to the remaining time in the timestamp, but we rule out the equality case because for this example we do not want a timeout straight away.

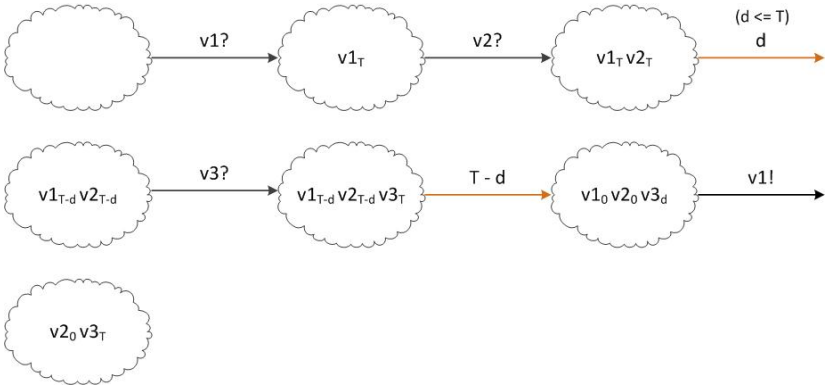


Figure 4.5: An example of a general timed list performing input, delay and output.

Let's say that at this point a new element is input as per the transition

$$\langle \vec{v}_2 \rangle_{T-d} \frown \langle \vec{v}_1 \rangle_{T-d} \frown [] \xrightarrow{\langle \vec{v}_3 \rangle ?} \langle \vec{v}_3 \rangle_T \frown \langle \vec{v}_2 \rangle_{T-d} \frown \langle \vec{v}_1 \rangle_{T-d} \frown []$$

A further delay is then possible causing both elements to time out:

$$\langle \vec{v}_3 \rangle_T \frown \langle \vec{v}_2 \rangle_{T-d} \frown \langle \vec{v}_1 \rangle_{T-d} \frown [] \xrightarrow{T-d} \langle \vec{v}_3 \rangle_d \frown \langle \vec{v}_2 \rangle_0 \frown \langle \vec{v}_1 \rangle_0 \frown []$$

Now the two initial elements have timed out i.e. their timestamps have reached zero, and thus by the rules FWD-HD and FWD they are ready for output. For arguments sake, let's say the first element to be input is output:

$$\langle \vec{v}_3 \rangle_d \frown \langle \vec{v}_2 \rangle_0 \frown \langle \vec{v}_1 \rangle_0 \frown [] \xrightarrow{\vec{v}_1 !} \langle \vec{v}_3 \rangle_d \frown \langle \vec{v}_2 \rangle_0 \frown []$$

This example has demonstrated all the features of timed lists: adding an element to the list via an input, allowing time to pass and elements to time out, and outputting timed out elements from the list. ▲

Example 4.17 (Input List Example)**.** Figure 4.6 is a depiction of the behaviour of a typical input list. For this example, we again begin with an empty list and sequentially input two elements as per the following trace

$$[] \xrightarrow{\langle \vec{v}_1 \rangle ?} \langle \vec{v}_1 \rangle_0 \frown [] \xrightarrow{\langle \vec{v}_2 \rangle ?} \langle \vec{v}_2 \rangle_0 \frown \langle \vec{v}_1 \rangle_0 \frown []$$

This is exactly the behaviour of a timed list whose time bound T is 0, as is the case for the input list. Now, since the timestamp on the elements begins at 0, they are initially already timed out and ready for output. Let's say one of the elements is output

$$\langle \vec{v}_2 \rangle_0 \frown \langle \vec{v}_1 \rangle_0 \frown [] \xrightarrow{\vec{v}_1 !} \langle \vec{v}_2 \rangle_0 \frown []$$

One element remains in the list, ready for output. Suppose another element is added to the list

$$\langle \vec{v}_2 \rangle_0 \frown [] \xrightarrow{\langle \vec{v}_3 \rangle ?} \langle \vec{v}_3 \rangle_0 \frown \langle \vec{v}_2 \rangle_0 \frown []$$

The list once again contains two elements that are both ready for output. However, according to the rule DEL-IN, these elements need not be output; they may be simply dropped from the list as some delay d transpires, modelling message loss:

$$\langle \vec{v}_3 \rangle_0 \frown \langle \vec{v}_2 \rangle_0 \frown [] \xrightarrow{d} []$$

This last rule is unique to input lists, allowing such a list to drop timed out elements as it delays, whereas a general timed list would be forced to output them before time could pass. ▲

Example 4.18 (Notification List Example)**.** Figure 4.7 portrays some standard behaviour of a notification list. Now let us look at an example of a notification list. The two rules for notification lists that extend the standard

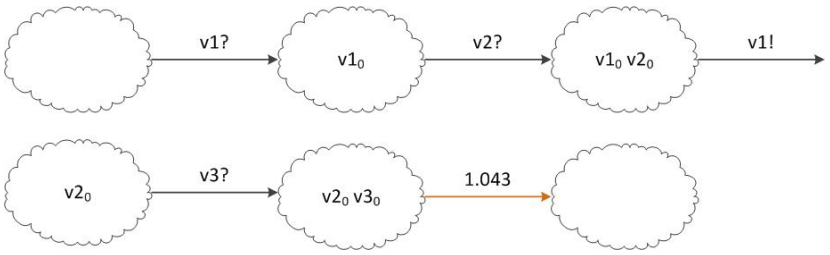


Figure 4.6: An example of an input list. Notice the delay behaviour by which all messages are lost.

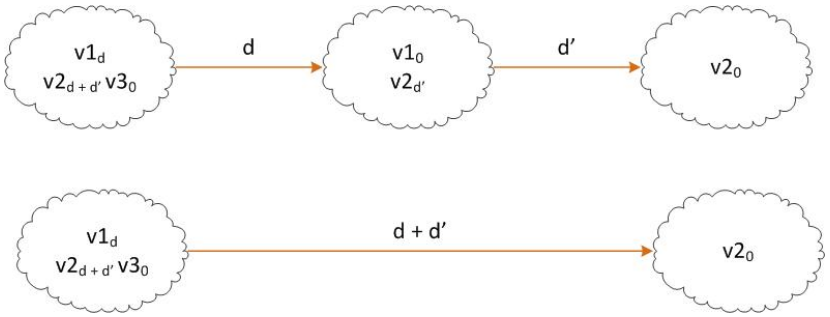


Figure 4.7: An example of a notification list. Notice the drastic delay behaviour by which all messages are lost.

timed lists semantics are DEL-NOTIF-ADD and DEL-NOTIF-DROP. The rule DEL-NOTIF-DROP says that if an element at the head of the list has timed out, it may be dropped from the list to allow time to pass for the rest of the list. On inspection the rule in fact implies that all elements that have timed out are dropped from the list so that time can pass. This is similar to the dropping of messages by an input list. For an example of this rule consider the transition

$$\langle \vec{v}_1 \rangle_d \frown \langle \vec{v}_2 \rangle_{d+d'} \frown \langle \vec{v}_3 \rangle_0 \frown [] \xrightarrow{d} \langle \vec{v}_1 \rangle_0 \frown \langle \vec{v}_2 \rangle_{d'} \frown []$$

A delay of d occurs causing the element $\langle \vec{v}_3 \rangle_0$ to be dropped while the timestamps of the other elements diminish; in particular, the element $\langle \vec{v}_1 \rangle_d$ times out. Now, once again a delay is possible, this time with the freshly timed out process being dropped

$$\langle \vec{v}_1 \rangle_0 \frown \langle \vec{v}_2 \rangle_{d'} \frown [] \xrightarrow{d'} \langle \vec{v}_2 \rangle_0 \frown []$$

Now let us consider the rule DEL-NOTIF-ADD. What it says is that if two sequential delays are possible, they may be added together, with the intermediate process being left out of the resulting transition. We have already seen earlier in this example that

$$\langle \vec{v}_1 \rangle_d \frown \langle \vec{v}_2 \rangle_{d+d'} \frown \langle \vec{v}_3 \rangle_0 \frown [] \xrightarrow{d} \langle \vec{v}_1 \rangle_0 \frown \langle \vec{v}_2 \rangle_{d'} \frown [] \xrightarrow{d'} \langle \vec{v}_2 \rangle_0 \frown []$$

Since this constitutes two sequential transitions, they may be joined, and the following transition is valid

$$\langle \vec{v}_1 \rangle_d \frown \langle \vec{v}_2 \rangle_{d+d'} \frown \langle \vec{v}_3 \rangle_0 \frown [] \xrightarrow{d+d'} \langle \vec{v}_2 \rangle_0 \frown []$$

Note that without the rule for adding delays, it would be impossible to infer this transition. ▲

4.3 Mode State Language

The mode state language provides a simple interface which records the current mode of an entity along with providing various functionality to allow an external observer to initiate a mode change, read the current mode,

abort a change in mode, or finalise a mode change. The actual operational details of what happens during a mode change are ignored; at this level of abstraction all that is considered is the time it takes to transition between any two modes, and whether such a transition is even possible at all. As such, the mode state language has underlying it two mathematical objects: a partial function tt and a relation $\rightsquigarrow$.

The relation $\rightsquigarrow$ precisely encodes the domain of tt. In other words, $tt(m,m')$ is defined for two modes m and m' precisely when $m \rightsquigarrow m'$.

The partial function tt maps pairs of modes to elements in the time domain. If defined, the meaning of $tt(m,m')$ is the maximum[3] time it takes to alter the state of an entity from mode m to m'.

Note that the mode transition time abstracts from possible details regarding the physical state of the entity. For example, if modes were ranges of speed, it is clear that a worst-case lower bound on the transition time between modes would be dependent on the maximum acceleration. To be specific, imagine the two modes (speed ranges) under consideration are $m = [0,10)$ and $m' = [10,20)$ respectively. Then when in m, it is possible that an entity is travelling at a speed of 0. Hence the minimum time in this worst case scenario that it will take to get to m', which requires the entity to be travelling at a speed of at least 10, will be $\frac{10}{a}$ where a is the maximum acceleration possible.

Let us now begin our exposition of the mode state language. First, we look at the syntax, which is quite simple and given in Table 4.12. A mode state object is either a singe mode m, signifying that the corresponding entity is currently in that mode. We say that such an entity is in a stable mode, and refer to m as a stable mode state. Alternatively, a mode state may be a triple $\langle m,m',t\rangle$, which we will call a transitional mode state. In this case, m is the current mode of operation, m' is a new mode to which the entity would like to transition, called the pending mode or the next mode, and t is the time remaining until this transition becomes possible, which we will call the time guard or simply the guard. The basic behaviour of the mode state is that it remains in a stable state until a request is made to change mode. Then it enters the transitional state in which it is changing from the current mode to the new mode. Finally, when enough time has elapsed, the mode state object times out and is ready to enter the new mode. This is more or less the intended operation of the mode state object. However, there are some other subtle behaviours which are covered during the explanation of the mode state semantics.

Syntax
$\mathbf{K} ::= m \mid \langle m,m',t\rangle$

Table 4.12: Syntax of the MState language.

The judgements in the semantics are as follows:

- $\mathbf{K} \xrightarrow{c\langle m\rangle?} \mathbf{K}'$ means that the mode state $\mathbf{K}$ inputs the mode m on the channel c and evolves to $\mathbf{K}'$.
- $\mathbf{K} \xrightarrow{c\langle m\rangle!} \mathbf{K}'$ is similar to the above, except that the mode is output.
- $\mathbf{K} \xrightarrow{c!} \mathbf{K}'$ is an output on the channel c but no message is passed.
- $\mathbf{K} \xrightarrow{d} \mathbf{K}'$ means that the mode state $\mathbf{K}$ delays by an amount d and subsequently becomes $\mathbf{K}'$.

Let us now discuss the rules for the semantics of the mode state language.

The first two rules for delay are straightforward. DEL-SING allows for a stable mode state to delay arbitrarily and remain unchanged. A similar arbitrary delay is allowed for a timed out transitional mode state by the rule DEL-TOUT. Such a mode state is waiting for an external signal to finalise the mode change, and in the meanwhile it simply delays. The next delay rule, DEL-GUARD, allows a transitional mode state to delay

[3] Of course, strictly speaking, the maximum time between mode changes is possibly infinite, since an entity can conceivably remain in the original mode indefinitely. The assumption here is that the entity in question is attempting a mode change.

by a certain amount of time and subtract this time from the guard. The final delay rule, TIMEOUT, allows a transitional mode state to delay by an amount beyond the time guard t. This rule is introduced to ensure certain desirable language properties, such as the property that a mode state can always delay by any amount of time.

The rule INIT allows a stable mode state m to input a mode update m' and become a transitional mode state. Note that the channel *chanNext* is used to receive mode updates. The modes in the resultant transitional mode state are, naturally enough, m and m', while the time is $tt(m,m')$, the maximum mode transition time between m and m', discussed above. Essentially, the transition from m to m' is guarded until this time elapses, at which point we say the mode state has timed out. Once timed out, a mode state is then ready to accept a request to switch modes. This is captured by the rule SWITCH, which allows a timed out mode state to input on the channel *chanCurr*, after which the pending mode becomes the current mode and the mode state stabilises.

One behaviour not yet discussed is the ability for a mode state to abort. If a mode state is in a transitional phase, then at any time it can be signalled to abort and return to the original stable state. The rule ABORT governs this behaviour. Note that the signal to abort is an input on the channel *chanStable*. This differs from a mode switch, which requires an input on *chanCurr*. Also note that, unlike the rule SWITCH, this rule does not require the transitional mode state to be timed out. The abort can happen at *any* time during the transitional phase.

The remaining rules allow an external observer of the mode state to gain information about it. The rules OUT-CURR-SING and OUT-CURR-TRANS allow a mode state to output its current mode. The rule OUT-NEXT allows a mode state to output its next mode. The rule OUT-STABLE allows a stable mode state to signal that it is stable. This is useful for an external observer who might need to have information regarding the stability of the mode state in order to progress. An interesting consequence of adding this polling for stability is that a mode state is always capable of outputting either on this channel or on *chanNext*; this fact is exploited by the protocol process.d

Semantics	
DEL-SING	$\dfrac{}{m \xrightarrow{d} m}$
DEL-TOUT	$\dfrac{}{\langle m,m',0\rangle \xrightarrow{d} \langle m,m',0\rangle}$
DEL-GUARD	$\dfrac{d \leq t}{\langle m,m',t\rangle \xrightarrow{d} \langle m,m',t-d\rangle}$
TIMEOUT	$\dfrac{t < d}{\langle m,m',t\rangle \xrightarrow{d} \langle m,m',0\rangle}$
INIT	$\dfrac{m \rightsquigarrow m'}{m \xrightarrow{chanNext\langle m'\rangle ?} \langle m,m',tt(m,m')\rangle}$
SWITCH	$\dfrac{}{\langle m,m',0\rangle \xrightarrow{chanCurr?} m'}$
ABORT	$\dfrac{}{\langle m,m',t\rangle \xrightarrow{chanStable?} m}$
OUT-CURR-SING	$\dfrac{}{m \xrightarrow{chanCurr\langle m\rangle !} m}$

OUT-CURR-TRANS	$\dfrac{}{\langle m, m', t\rangle \xrightarrow{chanCurr\langle m\rangle!} \langle m, m', t\rangle}$
OUT-NEXT	$\dfrac{}{\langle m, m', t\rangle \xrightarrow{chanNext\langle m'\rangle!} \langle m, m', t\rangle}$
OUT-STABLE	$\dfrac{}{m \xrightarrow{chanStable!} m}$

Table 4.13: Semantics of the Mode State language.

4.4 Tying it all together: Entity Network Language

Recall from the beginning of this chapter, particularly Figure 4.1, that the overall design of the language consists of point-to-point "inner" languages contained within a broadcast "outer" language. It is now time to present the outer broadcast language. Technically speaking, this is in fact two languages: one for single entities and another for networks of entities. However for simplicity they can both be considered together here as constituting an outer language.

The type of communication is open broadcast of values over a shared communication space. Unlike the inner point-to-point languages, there are no channels. The values for broadcast are vectors of base values $\vec{v}$ augmented with position l and coverage r information. We write such a value as $\langle \vec{v} \rangle_r^l$ and adopt the convention that the symbol w ranges over all such values:

$$w ::= \langle \vec{v} \rangle_r^l$$

These broadcasts may be either inputted, output or ignored, the meaning of which is explained later.

Networks are syntactically modelled as lists of entities, an idea taken from [Delzanno et al., 2010], though the order of entities lists is irrelevant beyond a means of identification. We allow for empty networks, i.e. empty lists, but these do not have any semantic behaviour, and are simply admitted to simplify modelling. Entities, usually denoted by **E**, are built of terms in the lower level languages already discussed. Every entity includes within it:

- A software process **P**.
- A position l.
- An interface component **I**.
- A mode state component **K**.

Many of the rules apply to single entities only and are given in a separate table to those applying to composite networks. These entity-level rules serve to combine the heterogeneous semantics of the three components **P**, **I** and **K** into a higher level semantics that makes sense.

Syntax
$\mathbf{N} ::= [] \mid \mathbf{E} \frown \mathbf{N}$ $\mathbf{E} ::= \lhd \mathbf{P}, l, \mathbf{I}, \mathbf{K} \rhd$

Table 4.14: Syntax of networks and entities.

The judgements in the upcoming semantics are as follows.

- Delay: $\mathbf{E} \xrightarrow{d} \mathbf{E}'$.
- Output: $\mathbf{E} \xrightarrow{w!} \mathbf{E}'$.
- Input: $\mathbf{E} \xrightarrow{w?} \mathbf{E}'$.
- Ignore: $\mathbf{E} \xrightarrow{w:} \mathbf{E}'$.
- Silent: $\mathbf{E} \xrightarrow{\tau} \mathbf{E}'$.

The need for an ignore relation is a consequence of the fact that broadcast should be non-blocking. While point to point calculi require parallel processes to synchronise input with output, and block otherwise, broadcast processes are allowed to ignore a broadcast whenever it cannot be inputted, allowing the outputting process to progress i.e. not blocking it. This resonates with real-world behaviour; it is absurd to imagine an entity blocked from broadcasting a message cross a wireless network just because nobody is listening.

The structural operational semantics of single entities build upon the lower level semantics of the software, interface and mode state languages, allowing the various components of an entity to communicate with each other over distinguished channels. Before delving straight into an explanation of each of the individual laws, it is worth taking one more look at the overall picture of how the sub-components of an entity are connected together via channels. To this end, we refer back to Figure 4.1, which depicts this interconnection of components. We now discuss this diagram in reasonable detail, with the intention of developing a basis for understanding the upcoming laws. We structure this discussion by analysing each of the channels individually:

- The channel *chanPos* allows a software component to read the current position. Hence any input action of the software process on this channel should result in the current position of the entity in question being consumed by said process.
- The channels *chanCurr*, *chanNext* and *chanStable* all allow intercommunication between the software process **P** and the mode state component **K**. *chanCurr* allows the **P** to read the current mode and to switch the current mode to the next mode. *chanNext* allows **P** to read the pending mode and to initiate a mode transition. Finally, the channel *chanStable* allows the software process to both test the mode state for stability and to abort the existing mode transition.
- The channels *chanInP*, *chanOutP* and *chanAN* allow for inter-communication between the software component **P** and the interface component **I**. *chanOutP* allows outgoing messages to be sent from **P** and buffered by **I** for eventual broadcast. Incoming messages from the environment which have already been buffered by **I** are sent via *chanInP* to **P** for consumption. The channel *chanAN* allows **I** to send notification messages back to **P**.
- Finally, then channel *chanIOE* connects the interface component **I** to the external environment. When a broadcast message is received by an entity, **I** consumes the message via this channel. Also, when **I** outputs a message on this channel, the message is broadcast to the environment. It is over this channel that the point to point and broadcast semantics are linked up.

Now it is time to explain the structural operational laws for single entities. We start with the first rule PROC-TAU, which lifts a τ transition of the software process to the entity level. The rules IO-PROC-INTER, IO-INTER-PROC and NOTIF deal with intercommunication between the software component and the interface. IO-PROC-INTER deals with the case where the software process outputs a message which is intercepted and buffered by the interface. IO-INTER-PROC deals with the case where the software process consumes a buffered message that the interface is willing to output. NOTIF models delivery notification: the interface component sends a message to the software component over the channel *chanAN*. In all three cases, the resultant transition is a τ action since, even though the message passing involved is external to both the software process and the

interface, it is still internal to the entity and so should be hidden from the environment just as a synchronisation is hidden in regular point to point calculi.

The next six rules all deal with communication between the mode state and the software process. RD-STBL allows the software component to input on the channel *chanStable* when the corresponding mode state can output on that channel, which is the case exactly when the mode state is stable. The rules WRITE-STBL, CURR-WRITE, CURR-READ, NEXT-WRITE and NEXT-READ all allow for communication between the software process and the mode state over the channels *chanStable*, *chanCurr* and *chanNext*. The rules for reading, namely CURR-READ and NEXT-READ allow the software process to read the value of the current/next mode respectively. The write rules allow the software process to change the mode state. The nature of this change is dealt with at the lower level semantics. Again, all synchronisations are lifted to silent τ actions at the entity level.

There is only one rule governing communication over the channel *chanPos*, namely POS-READ. This rule allows the software process to consume the current value of the position l. Notice that there is no sub-language for the position component of an entity. This is the case because the role of the position component is too simple to warrant is own language: POS-READ is the only law relating to it. It is also interesting to note that a read by the software component on the channel *chanPos* will always be non-blocking in the context of an entity i.e. the rule POS-READ always allows a willing software process to input on the channel *chanPos*. This is in contrast, for example, to the situation with the channel *chanStable*, which only allows an input if the mode state is stable.

The rules IGNORE, IN and OUT all govern how an entity is to ignore, input or output a message respectively. The rules IGNORE and IN are of particular interest as they each include a novel premise asserting a fact about distances. The rule IGNORE says that whenever the message is out of range, then it is simply ignored and the entity remains unchanged after the ignore action takes place. The assertion that the message is out of range is handled in the premise of this rule. It simply states that the distance between the position of this entity and the position of the sender entity is greater than the delivery radius. This is how coverage is encoded in the language. The first premise of the rule IN covers the complementary case in which the message is within range. In this case, the message is consumed by the interface component along the channel *chanIOE*. The rule OUT lifts an output on the channel *chanIOE* by the interface component to an output broadcast at the entity level.

DEL-ENT states that whenever all sub-components of an entity can delay by some amount d, then so can the whole entity. However, there is also an additional clause $noSyncEnt(\mathbf{P},\mathbf{I},\mathbf{K},d)$ which asserts that there is no synchronisation between the various components at any point during the delay. This ensures that whenever synchronisation *is* possible, it will progress. The actual details and formal definition of this predicate follow at a later stage, see Definition 4.23.

Remark 4.19 (Entity Movement)**.** A novel feature of this language that sets it apart from other languages in the literature is the way in which mobility for entities is modelled. Most existing languages are either hybrid languages for the specification of exact continuous dynamics via differential equations, or they omit continuous behaviour altogether. A number of other languages, particularly those that have recently been developed, include a notion of mobility but it is an abstract sort of mobility e.g. a changing network topology modelled as a connectivity graph. The novelty of the mobility included in this calculus is that it is highly non-deterministic: the exact physical dynamics of entities need not be specified beyond a maximum possible speed for an entity. The advantage of this is abstraction: the model is not bound to any particular rules for the movement of entities other than a maximum speed, which is a reasonable assumption.

Of course, further restrictions could be placed on the movement of entities in this language, as some movements do not resonate with real-world behaviour. For example, an entity that jumps from one position to another at maximum speed across one delay, and then jumps back to the initial position across another delay of equal length is absurd: it implies infinite acceleration. A “real” path would be continuous, obeying some sort of physical laws of motion. However, this over-inclusion of behaviours is not a problem: safety of a superset implies safety of the “sensible” subset. In any case, a refinement of Comhordú to include more system specific

detail would enforce laws leading to more realistic behaviour.

To recap, movement is modelled across a delay by the rule DEL-ENT. Each entity in the source network can "move" during the delay by jumping to another position in the derivative network. The size of the jump is bounded from above by the maximum speed of an entity s_{max} multiplied by the delay. Note that the jump in position doesn't necessarily model movement in a straight line. The exact trajectory during the delay is irrelevant at this level of abstraction. All that is modelled is the start and end positions of the entity. However, since there is no lower limit on the size of a delay, a sequence of delays could approximate a given trajectory as a series of jumps, to an arbitrary level of detail.

Figure 4.8 depicts entities in a network moving across a delay. A circle around each entity represents the upper bound $s_{max} * d$ on the displacement: an entity cannot move outside this circle within the given time frame due to the assumed maximum bounding speed. Note that the circles are not actually objects within the network but are merely imaginary lines constructed to convey the bound. Similarly, the old position of entities in the derivative network is shown, but there is no explicit history of such contained in the network; in fact a network history need not be unique. An arrow labelled with d linking the two states represents the delay transition. ▲

Remark 4.20 (Entity Overlap)**.** One seemingly unrealistic artefact of this language is the possibility that entities might occupy the same point in space. Clearly this appears absurd: the region of space occupied by any physical object usually cannot overlap with that of another object. However, this feature is not problematic at the level of generality of this work. It is left to the system designer to encode collision freedom according to the *minimum distance of compatibility* function, or alternatively to use another protocol to achieve this.

Furthermore, there are scenarios where the "same point" might not actually mean the same point in space. For example imagine a system of aerial vehicles operating in planes, where planes represent modes and only two dimensional position is recorded. In such a system, entities could be directly over each other and so occupying the "same point" in terms of their two dimensional positions, but would be physically separate because of the vertical separation. A similar example would be cars on a road with access to a "hard shoulder", where only one dimensional position is recorded for each vehicle. The following conditions roughly characterise this idea in general, and are sufficient for Comhordú to be applicable to the achievement of collision freedom in a system.

- Space can be discretely divided into sub-spaces representing modes with at least one fail-safe mode.
- An entity at any point in space can always reach an unoccupied region of a fail-safe space in a bounded time along an unobstructed path. This is quite a strong condition in that it assumes that that there is always a trajectory through space and time that an entity can take that will be completely free of obstacles.
- Entities that are fully in a fail-safe space do not move, except to leave that space. The reason for the qualification "fully in" is that as an entity enters/exits a fail-safe space it inevitably must be moving.

While the considerations here are of interest in relation to the applicability of Comhordú, further pursuit of this topic is beyond the scope of this work. ▲

The final rule in these single entity semantics is TEST. This rule allows for the non-deterministic initiation of a change of mode whenever the mode state is willing to accept such a mode change. At the mode state level, this initiation is in the form of an input action, but at the entity level this is lifted to a silent τ action. The reason for this law is to abstract from the exact nature of the "driver" or "pilot" of an entity and instead encompass all possible "driver" behaviours via non-determinism. It is worth noting that such a law could lead to zeno behaviours, i.e. infinite action in bounded time, depending on the nature of the software process. However, these behaviours can be ignored for the purposes of this analysis, since they do not affect safety. For a short discussion on these behaviours, please see Remark 4.21.

Remark 4.21 (Zeno Behaviours at the Entity level)**.** Recall from Example 4.12 that zeno behaviour is possible for a software process. One question that seems relevant at this point is, can an *entity* exhibit zeno behaviour? The answer to this is quite simply yes: it is straightforward to show that zeno behaviour at the software level

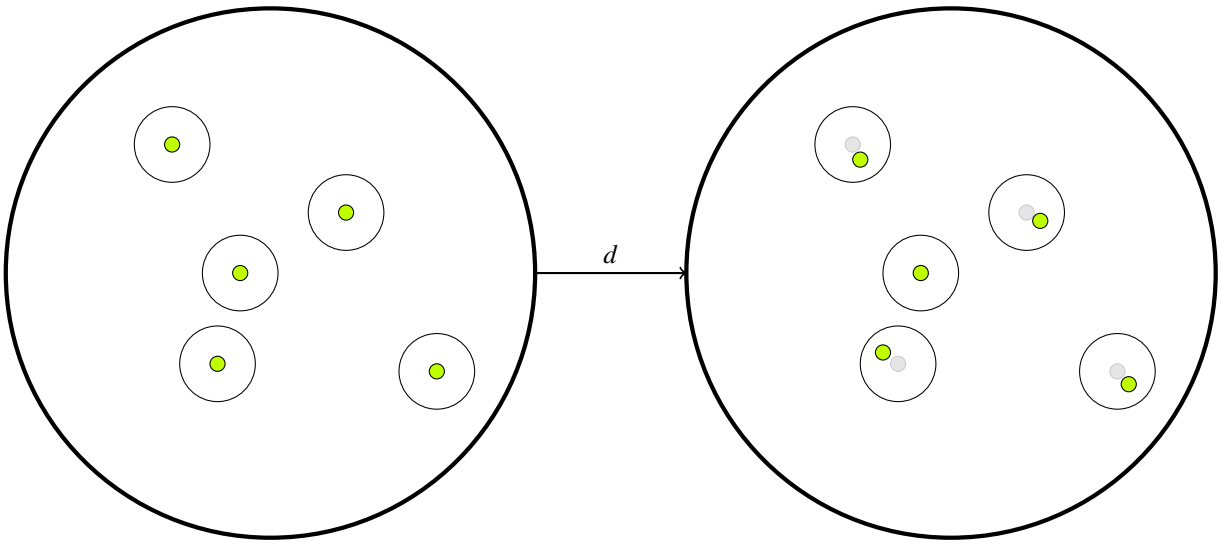


Figure 4.8: Movement of entities across a delay. Entities are shown as dots. Circles around each entity signify the area to which an entity can move across a delay. The old positions of entities are shown faded in the derivative state. Note that not all entities *have* to change position across the delay i.e. a displacement of 0 is possible.

results in that at the entity level, and the former has already been shown. This relationship can be seen by the rule PROC-TAU, which allows a τ transition of the software process to be "lifted", so to speak, to the entity level. The formal definition of zenoness at the entity level is much that same as that at the software level, given in Definition 4.11 and so it is omitted here. It still remains open whether zenoness is problematic for this model of Comhordú. This is discussed in Section 8.2.4. ▲

Before arriving at the formal definition of the semantics, a predicate for no-synchronisation *noSyncEnt* must be formulated. This is similar to the no-synchronisation predicate for processes given in Definition 4.4; the idea is to ensure maximal progress i.e. the impossibility of delay while synchronisations on input/output half actions are possible. Differing from the software case though, the predicate *noSyncEnt* does not fall into the same problem of self reference. That is, the semantics in which it is used- the entity semantics- are different to those upon which it is formulated. This predicate is formally defined in Definition 4.23 in terms of an auxiliary predicate *noComm*which is defined in Definition 4.22.

Definition 4.22 (*noComm*)**.** $noComm(\mathbf{P},\mathbf{I},\mathbf{K})$ asserts that no communication is currently possible between the software process $\mathbf{P}$ and either $\mathbf{I}$ or $\mathbf{K}$ i.e. they cannot perform complementary I/O actions. The predicate also asserts that $\mathbf{P}$ is incapable of an input on the position channel *chanPos*, which is necessary since such inputs can lead to τ actions due to the rule POS-READ. In the following formal definition, $\mathbf{P} \xrightarrow{c\langle\vec{v}\rangle?}$ means that the process $\mathbf{P}$ is capable of performing the transition $\xrightarrow{c\langle\vec{v}\rangle?}$ to *some* derivative process. This notation also applies to $\mathbf{K}$ and $\mathbf{I}$.

$$
\begin{aligned}
noComm(\mathbf{P},\mathbf{I},\mathbf{K}) \stackrel{\text{def}}{=} \forall\, c,\vec{v},& \\
&\mathbf{P} \xrightarrow{c\langle\vec{v}\rangle?} \Rightarrow \neg(\mathbf{I} \xrightarrow{c\langle\vec{v}\rangle!} \vee \mathbf{K} \xrightarrow{c\langle\vec{v}\rangle!}) \wedge \\
&\mathbf{P} \xrightarrow{c\langle\vec{v}\rangle!} \Rightarrow \neg(\mathbf{I} \xrightarrow{c\langle\vec{v}\rangle?} \vee \mathbf{K} \xrightarrow{c\langle\vec{v}\rangle?}) \wedge \\
&\forall l, \neg\mathbf{P} \xrightarrow{chanPos\langle l\rangle?}
\end{aligned}
$$

■

Definition 4.23 (*noSyncEnt*)**.** $noSyncEnt(\mathbf{P},\mathbf{I},\mathbf{K},d)$ asserts that no synchronisation is possible between the

sub-components of an entity during a delay. The predicate is defined by:

$$\begin{aligned} noSyncEnt(\mathbf{P},\mathbf{I},\mathbf{K},d) \stackrel{\text{def}}{=} noComm(\mathbf{P},\mathbf{I},\mathbf{K}) \wedge \forall\, d',\mathbf{P}',\mathbf{I}',\mathbf{K}', \\ d' < d \wedge \mathbf{P} \xrightarrow{d'} \mathbf{P}' \wedge \mathbf{I} \xrightarrow{d'} \mathbf{I}' \wedge \mathbf{K} \xrightarrow{d'} \mathbf{K}' \\ \Rightarrow noComm(\mathbf{P}',\mathbf{I}',\mathbf{K}') \end{aligned}$$

What this says is that for all delay derivatives of the components **P**, **I** and **K** within the given time d, these processes are incapable of inter-communication. This condition is itself defined above in Definition 4.22. ■

Remark 4.24 (Including the predicate *noSyncEnt*: a problem?)**.** In the presentation of the software language, a predicate *noSync*is considered in an attempt to give the language the property of maximal progress. This predicate is quickly shown to run into the problem of self-reference: its the definition relies on the software delay semantics which in turn would rely upon it if it were used. Hence it is abandoned in favour of an alternative characterisation, namely *Sort*.

The question then that arises here is, does the predicate *noSyncEnt* fall into the same difficulty? The answer is no, and the reason why is quite obvious. The formulation of *noSyncEnt* is based upon the delay behaviour of the *underlying* semantics of the software, interface and mode state languages. It does not mention the delay behaviour of entities. Therefore, it is perfectly reasonable to include this predicate in the definition of the delay semantics for entities, as has been done. ▲

Semantics	
PROC-TAU	$\dfrac{\mathbf{P} \xrightarrow{\tau} \mathbf{P}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I},\mathbf{K}\rhd}$
IO-PROC-INTER	$\dfrac{\mathbf{P} \xrightarrow{chanOutP\langle\vec{v}\rangle!} \mathbf{P}' \qquad \mathbf{I} \xrightarrow{chanOutP\langle\vec{v}\rangle?} \mathbf{I}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I}',\mathbf{K}\rhd}$
IO-INTER-PROC	$\dfrac{\mathbf{P} \xrightarrow{chanInP\langle\vec{v}\rangle?} \mathbf{P}' \qquad \mathbf{I} \xrightarrow{chanInP\langle\vec{v}\rangle!} \mathbf{I}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I}',\mathbf{K}\rhd}$
NOTIF	$\dfrac{\mathbf{P} \xrightarrow{chanAN\langle\vec{v}\rangle?} \mathbf{P}' \qquad \mathbf{I} \xrightarrow{chanAN\langle\vec{v}\rangle!} \mathbf{I}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I}',\mathbf{K}\rhd}$
RD-STBL	$\dfrac{\mathbf{P} \xrightarrow{chanStable?} \mathbf{P}' \qquad \mathbf{K} \xrightarrow{chanStable!} \mathbf{K}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I},\mathbf{K}'\rhd}$
WRITE-STBL	$\dfrac{\mathbf{P} \xrightarrow{chanStable!} \mathbf{P}' \qquad \mathbf{K} \xrightarrow{chanStable?} \mathbf{K}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I},\mathbf{K}'\rhd}$
CURR-WRITE	$\dfrac{\mathbf{P} \xrightarrow{chanCurr!} \mathbf{P}' \qquad \mathbf{K} \xrightarrow{chanCurr?} \mathbf{K}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I},\mathbf{K}'\rhd}$
CURR-READ	$\dfrac{\mathbf{P} \xrightarrow{chanCurr\langle m\rangle?} \mathbf{P}' \qquad \mathbf{K} \xrightarrow{chanCurr\langle m\rangle!} \mathbf{K}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I},\mathbf{K}'\rhd}$
NEXT-WRITE	$\dfrac{\mathbf{P} \xrightarrow{chanNext\langle m\rangle!} \mathbf{P}' \qquad \mathbf{K} \xrightarrow{chanNext\langle m\rangle?} \mathbf{K}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I},\mathbf{K}'\rhd}$
NEXT-READ	$\dfrac{\mathbf{P} \xrightarrow{chanNext\langle m\rangle?} \mathbf{P}' \qquad \mathbf{K} \xrightarrow{chanNext\langle m\rangle!} \mathbf{K}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I},\mathbf{K}'\rhd}$

POS-READ	$\dfrac{\mathbf{P} \xrightarrow{chanPos\langle l\rangle?} \mathbf{P}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P}',l,\mathbf{I},\mathbf{K}\rhd}$
IGNORE	$\dfrac{|l-l'|>r}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\langle\vec{v}\rangle_r^{l'}:} \lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd}$
IN	$\dfrac{|l-l'|\leq r \qquad \mathbf{I} \xrightarrow{chanIOE\langle\vec{v}\rangle?} \mathbf{I}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\langle\vec{v}\rangle_r^{l'}?} \lhd\mathbf{P},l,\mathbf{I}',\mathbf{K}\rhd}$
OUT	$\dfrac{\mathbf{I} \xrightarrow{chanIOE\langle\vec{v}\rangle_r!} \mathbf{I}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\langle\vec{v}\rangle_r^{l}!} \lhd\mathbf{P},l,\mathbf{I}',\mathbf{K}\rhd}$
DEL-ENT	$\dfrac{\mathbf{P} \xrightarrow{d} \mathbf{P}' \quad \mathbf{I} \xrightarrow{d} \mathbf{I}' \quad \mathbf{K} \xrightarrow{d} \mathbf{K}' \quad noSyncEnt(\mathbf{P},\mathbf{I},\mathbf{K},d) \quad |\Delta l| \leq s_{max}.d}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{d} \lhd\mathbf{P}',l+\Delta l,\mathbf{I}',\mathbf{K}'\rhd}$
TEST	$\dfrac{\mathbf{K} \xrightarrow{chanNext\langle m\rangle?} \mathbf{K}'}{\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd \xrightarrow{\tau} \lhd\mathbf{P},l,\mathbf{I},\mathbf{K}'\rhd}$

Table 4.15: Semantics of single entities.

4.4.1 Rules for Networks of Entities

We must still give an overall semantics for networks of entities. Having already covered the semantics of single entities, the network semantics are fairly straightforward, dealing mostly with communications between entities. The semantics for networks of entities are given in Table 4.16; an auxiliary relation is also defined in . It is now useful to give some explanation of these.

Let us begin with the rules for ignore: IGNORE-NET and IG-EMP. The former says that a network can ignore a value if all of its constituent entities can ignore that value. Like many of the rules to come, this rule works by induction, stripping off entities from the head of the list and recursively drilling into the tail. The special case is that of the empty network, which is covered by the rule IG-EMP, saying that the empty network can ignore any value.

The remaining discrete rules cover output, input and τ actions. The output rules are OUT-HEAD and OUT-TAIL. The former says that if the first entity in a network can output a value, and the remaining network accepts this value, then the network can output this value with index 0. The latter says that if a network can output a value with a certain index, and an entity can accept this value, then the new network formed by appending the entity to the start of that network can output the value, with the index incremented. The rules IN-HEAD and IN-TAIL are almost analogous to the rules for output, except there are no indices involved. The rules TAU-HEAD and TAU-TAIL say that if an entity in the network can perform a silent action, then the whole network can perform that action, while only the entity in question evolves and the rest of the network remains the same.

The semantics for delay are captured by the rules DEL-EMP and DEL-NET, the former allowing the empty network to delay by any amount, the latter allowing a non-empty network to delay iff the head and tail can both delay.

Remark 4.25 (Networks as Lists)**.** It may come to mind that the choice of a list is strange to model a network of entities. After all, in reality these entities are distributed in space and don't form a natural order. One may then propose sets as a more apt data structure to model networks. However, there are a number of advantages of choosing lists over sets:

- Lists are easier to model in a programming language.

- Lists allow multiple copies of the same entity to be present. Of course, so do multisets, but these will probably be modelled as lists anyway in practice.
- Lists allow one to keep track of entities via an index. Entities change as networks evolve, but their place in the list doesn't; the semantics ensures this. Hence in a network trace it makes sense to talk about "entity i" over multiple states, even though the actual entity at position i may have changes over these states. Without this it would be difficult to link an entity in one state to an entity in another state and assert that one evolved into the other.
- Entity indices, which naturally follow from the list, allow us to identify the entity in a network responsible for an output broadcast, which will be a useful detail in later proofs.

For these reasons, lists are the choice of data structure to model networks. However, it is important to note that the ordering in the list doesn't correspond to any ordering on the entities, other than allowing them to be uniquely identified. ▲

The output action in the upcoming semantics of Table 4.16 differs from that of entities in that it is augmented with an identifier i, indicating which entity in the network performed the broadcast. Other than this, the judgements in the network semantics are analogous to those of the single entity semantics.

- Delay: $\mathbf{N} \xrightarrow{d} \mathbf{N}'$.
- Output: $\mathbf{N} \xrightarrow{w!_i} \mathbf{N}'$.
- Input: $\mathbf{N} \xrightarrow{w?} \mathbf{N}'$.
- Ignore: $\mathbf{N} \xrightarrow{w:} \mathbf{N}'$.
- Silent: $\mathbf{N} \xrightarrow{\tau} \mathbf{N}'$.

Semantics			
IG-EMP	$\dfrac{}{[] \xrightarrow{w:} []}$	IGNORE-NET	$\dfrac{\mathbf{E} \xrightarrow{w:} \mathbf{E}' \qquad \mathbf{N} \xrightarrow{w:} \mathbf{N}'}{\mathbf{E} \frown \mathbf{N} \xrightarrow{w:} \mathbf{E}' \frown \mathbf{N}'}$
OUT-HEAD	$\dfrac{\mathbf{E} \xrightarrow{w!} \mathbf{E}' \qquad \mathbf{N} \xrightarrow{w?:} \mathbf{N}'}{\mathbf{E} \frown \mathbf{N} \xrightarrow{w!_0} \mathbf{E}' \frown \mathbf{N}'}$	OUT-TAIL	$\dfrac{\mathbf{E} \xrightarrow{w?:} \mathbf{E}' \qquad \mathbf{N} \xrightarrow{w!_i} \mathbf{N}'}{\mathbf{E} \frown \mathbf{N} \xrightarrow{w!_{i+1}} \mathbf{E}' \frown \mathbf{N}'}$
IN-HEAD	$\dfrac{\mathbf{E} \xrightarrow{w?} \mathbf{E}' \qquad \mathbf{N} \xrightarrow{w?:} \mathbf{N}'}{\mathbf{E} \frown \mathbf{N} \xrightarrow{w?} \mathbf{E}' \frown \mathbf{N}'}$	IN-TAIL	$\dfrac{\mathbf{E} \xrightarrow{w:} \mathbf{E}' \qquad \mathbf{N} \xrightarrow{w?} \mathbf{N}'}{\mathbf{E} \frown \mathbf{N} \xrightarrow{w?} \mathbf{E}' \frown \mathbf{N}'}$
TAU-HEAD	$\dfrac{\mathbf{E} \xrightarrow{\tau} \mathbf{E}'}{\mathbf{E} \frown \mathbf{N} \xrightarrow{\tau} \mathbf{E}' \frown \mathbf{N}}$	TAU-TAIL	$\dfrac{\mathbf{N} \xrightarrow{\tau} \mathbf{N}'}{\mathbf{E} \frown \mathbf{N} \xrightarrow{\tau} \mathbf{E} \frown \mathbf{N}'}$
DEL-EMP	$\dfrac{}{[] \xrightarrow{d} []}$	DEL-NET	$\dfrac{\mathbf{E} \xrightarrow{d} \mathbf{E}' \qquad \mathbf{N} \xrightarrow{d} \mathbf{N}'}{\mathbf{E} \frown \mathbf{N} \xrightarrow{d} \mathbf{E}' \frown \mathbf{N}'}$

Table 4.16: Semantics of networks of entities.

Notice the appearance of terms such as $\mathbf{N} \xrightarrow{w?:} \mathbf{N}'$. This represents an auxiliary relation upon which these semantics are based called the accept relation. This relation is simply the union of the ignore and input relations: a network can accept a value iff it can either input it or ignore it. Strictly speaking, the network semantics could be formulated without the accept relation but this approach would be unnecessarily verbose, requiring separate laws for the cases of input and ignore. This relation is formally defined in Table 4.17 with a small SOS. The term $\mathbf{N} \xrightarrow{w?:} \mathbf{N}'$ can be read as "the network $\mathbf{N}$ accepts the value w and evolves to become $\mathbf{N}'$".

Accept Relation			
ACC-IN	$\dfrac{\mathbf{N} \xrightarrow{w?} \mathbf{N}'}{\mathbf{N} \xrightarrow{w?:} \mathbf{N}'}$	ACC-IG	$\dfrac{\mathbf{N} \xrightarrow{w:} \mathbf{N}'}{\mathbf{N} \xrightarrow{w?:} \mathbf{N}'}$

Table 4.17: New accept relation is the union of the ignore and input relations, and simplifies the network semantics.

Chapter 5

Protocol

This chapter presents the Comhordú protocol, which is a term in the software fragment of the multi-tiered calculus that is developed in Chapter 4. The purpose of the protocol is to act as a software controller for entities to ensure that their behaviour is safe. To begin with, some motivations towards the development of a protocol are put forth in Section 5.1. Section 5.2 presents an informal description of the protocol. The formal protocol specification is introduced in Section 5.3. The chapter concludes with a summary of the protocol in Section 5.4.

5.1 Motivation

In order to motivate the need for a protocol, it is useful to show that without one there is a possibility of unsafe behaviour. Recall the definition of safety from Section 3.3.3. This roughly says that for all reachable states within a Comhordú system, there should be no two entities that are "too close together"; more formally it says that every pair of entities is separated by at least the minimum distance of compatibility for their respective modes. The question we want to ask now in relation to this is, can safety be violated in general according to the rules of the language? This question itself is in fact slightly ambiguous. A simple interpretation would be, are there any unsafe networks in the language? However, the answer to this is a trivial "yes". To construct such a network, simply choose two entities with certain modes and positions such that their distance falls short of the minimum distance of compatibility for the modes in question.

The existence of such trivially unsafe states does not offer much in and of itself. Just because such a state can be constructed, it doesn't mean that it is "reachable" in some sense. This then leads to the question of how reachability should be defined. An initial attempt at this is to say that a state is reachable if there is a sequence of actions and states leading from some safe state to this one. In other words, if some safe state is chosen, then all "future" states of this one according to the SOS may be called "reachable".

This now allows us to ask the question again- can safety be violated in general for some *reachable* state? The answer to this is not as trivial as before. It no longer suffices to choose an arbitrary unsafe state; it is now necessary to choose a safe state and show that from this an unsafe state can be reached. However, although this is no longer trivial, it is possible to demonstrate such a scenario without much difficulty. Consider a safe state in which an entity is just about to switch mode, where the *new* mode will cause an incompatibility once the switch takes place. It is easy to construct as the software component of this entity a process whose next action is to finalise the mode switch. Hence, it is possible for the switch to happen and safety to be violated. Thus we have demonstrated that safety is not a closed property under the transition relations: it is possible to arrive at an unsafe state from a safe one. This roughly demonstrates the need for a protocol, though there are a few more issues to untangle first.

Notice that for the previous example, it was necessary to choose the software component such that it exhibited certain behaviour in order to violate safety. The question then seems to be, with the right choice of software controller, can safety be maintained? Again, the answer to this is not straightforward, because the

question is not rigorous. In particular, we focus on the notion of "choice of software controller". Initially, it would appear that this means a certain term in the software language that should be contained within every entity as it software component. However, this interpretation immediately becomes absurd when a consequence is considered: it would mean that the software component would be incapable of evolution, frozen in a certain state, and thus for all intents and purposes useless.

The problem here is a mismatch between intuition and formalism. Intuitively, we associate a program with a piece of code that can be "in a certain state" as it is being "run". However, in this formalism, there is no separate notion of state other than a term itself. Program evolution then corresponds to the transformation of the term to a new term according to the transition rules as opposed to the term being fixed and some sort of program counter / memory being updated.

Thus we consider classes of states/terms rather than a single term. The question then becomes, does adherence to this class of process terms by all entities guarantee the preservation of safety? A related question would be, is this class of process terms closed under the transition relation? While this is somewhat in line with how the safety condition in the model is formulated, there is a simpler perspective on all of this.

Instead of considering *all* the safe states, we could consider only a subset which we call initial. We can then define reachability in a stricter way than before in terms of initial states rather than simply safe states. In the same vein as before then, we can ask two questions. First, is it possible for an initial state to eventually reach an unsafe state according to the structural operational semantics? Second, if all entities in an initial state contain a certain distinguished process, namely the Comhordú protocol, will safety be preserved for all possible futures of that state? If the answer to the first question is yes, then the need for Comhordú is justified. The answer to the second question constitutes the verification proof of Chapter 6.

Returning to the question of whether an initial state can reach an unsafe state, this is indeed the case. A scenario demonstrating this can be constructed by slightly more complex means than before, this time with two entities initiating and finalising mode switches to cause incompatibility. However, such a construction and discussion is omitted here as it would involve an unnecessarily complex exposition of the meaning of initial states, among other things. Rather, we simply accept its existence as motivation for the development of Comhordú.

An informal Venn-like diagram is given in Figure 5.1 consolidating many of the ideas discussed up until this point. Depicted conceptually is the set of states containing a subset of safe states, and a subset of these again which are the initial states. Two particular initial states s_1 and s_2 are shown, from which traces emerge, ending in $s_1{}'$ and $s_2{}'$ respectively. Traces are shown as sequences of states connected by arrows. In order to reduce clutter the action labels are omitted from the arrows, but they are assumed to exist.

All states along each trace are reachable; the sequence of states and transitions leading to a reachable state constitutes its proof of reachability. In the special case of an initial state, the trace in question is just the singleton trace containing the state itself. The state $s_2{}'$ is unsafe, representing the fact that some unsafe states are reachable.

A subset of the initial states, not shown in the diagram, called the initial protocol states are those that are "running the protocol" so to speak. Protocol correctness is defined in terms of these states: they should have the property that all traces emerging from them do not leave the set of safe states.

It is now time to describe the logistics of this protocol. Section 5.2 provides an informal description of how the protocol works, giving an insight into the nebula of ideas from which it was first created. Following this in Section 5.3 is the formal protocol specification. This adds rigour to the informal ideas and allows for some explorations of the protocol behaviour via traces.

5.2 Informal Protocol Description

Prior to a detailed formal presentation and analysis of the Comhordú protocol, let us first informally describe it at a high level here in order to portray the main ideas therein without getting lost in the detail that comes with

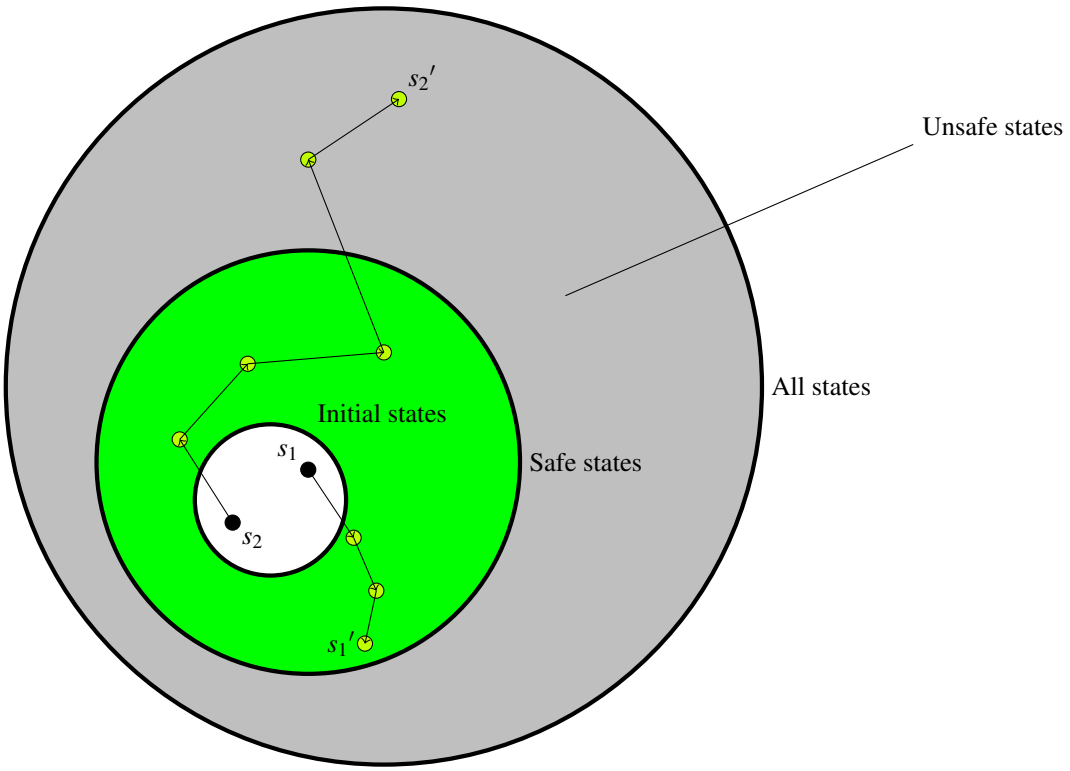


Figure 5.1: Safe, initial and reachable states. Some reachable states are safe, others aren't.

formalisation. The protocol works as follows. An entity **E** in a non-fail-safe mode m broadcasts its position l and its mode m to the surrounding entities in its environment. Each broadcast will have associated with it a coverage i.e. a range to which it was delivered.

The entity **E** will continue acting in mode m, unless one of two events happens: receipt of a message from another entity or degradation of coverage. In the case of receipt of a message from another entity, the received message is analysed in order to determine whether or not **E** and $\mathbf{E}'$ are at risk of becoming incompatible in the near future. If there is such a risk, then **E** will immediately begin to transition to some fail-safe mode $fSucc(m)$. Coverage degradation occurs when the broadcast range of the entity is no longer sufficient to guarantee safety. This event will also result in the entity transitioning to a fail-safe mode.

A couple of things from the previous explanation need further clarification, namely the notions of "sufficient coverage" and "at risk of becoming incompatible". These are defined in Definition 6.6 and Definition 5.5 respectively. Still, some motivation is needed before arriving at the definition of sufficient coverage. Let us step through some zone-based reasoning in order to provide such motivation. This zone based reasoning is based on similar reasoning from the original model, though the specifics are different.

The question that must be asked is, when is the coverage of a message sufficient to allow an entity to continue acting in its current mode? A coverage is sufficient if it is at least as large as some minimum coverage that is necessary to allow continued operation in the current mode. Figure 5.2 depicts this minimum sufficient coverage. The coverage is split into a number of zones. These zones are discussed in the following.

We define the region of an entity **E** as the area about **E** in which all other entities $\mathbf{E}'$ are guaranteed to have received its last message. At the time of delivery of a message, this is exactly equal to the coverage of the message. However, as time passes, entities outside the region can enter it and hence the region diminishes over time until the next message is delivered. Assuming a maximum absolute speed s_{max} for an entity, we get the upper bound $2 * s_{max} * \Delta * t$ on the amount the region has decreased within a time interval Δt. The factor of 2 here accounts for the fact that the maximum relative speed between entities is twice the maximum absolute speed of a single entity: it is the speed one entity travels with respect to the other when both are travelling at

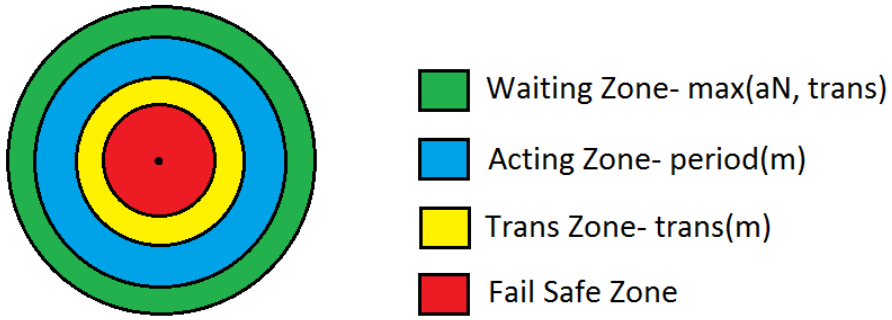


Figure 5.2: A minimum sufficient coverage split into its constituent zones.

maximum absolute speeds respectively in opposite directions[1]. As usual, we assume a worst-case shrinking of the region and hence relate the sizes of zones in the coverage to the times spent in these zones. The following describes each of the zones.

- The first zone, called the waiting zone, corresponds to the time in which **E** waits for its message to take effect in the environment around it. During this time, **E** also waits for a coverage notification message, informing it of the region to which the message in question was sent.
- The next zone is the acting zone. At this point, **E** knows that its environment is compatible with it and that its previous message coverage was sufficient. The duration of the acting zone must be at least $period(m)$, where m is the current mode of operation, since the next message will not come into effect until $period(m)$ time units.
- After the acting zone, there is a trans zone, which corresponds to the time an entity needs to take to transition to a fail-safe mode should this become necessary for reasons mentioned above.
- Finally, the innermost zone, the fail-safe zone, corresponds to the max-min distance of compatibility $d_{max}(m)$ for the mode m. The region must always be at least this size while **E** is still in mode m to ensure that all entities are compatible with it. We will now elaborate on all of this.

For the moment, let us assume that all entities that receive a message from a sender **E** will begin immediately transitioning to some fail-safe mode $fSucc(m)$. We assume an upper bound on the time it takes an entity in mode m to transition to a fail-safe mode. We define this time $trans(m)$ in Definition 5.2. From this, we can deduce the maximum time $trans$ it takes an entity in an arbitrary mode to reach a fail-safe mode. This is defined in Definition 5.3. In the worst case, it must be assumed that there is an entity that takes this amount of time to transition to a fail-safe mode. This implies that after the message propagation delay $msgLatency$, there must be a further delay of at least $trans$ before an entity can start acting.

Now, we know that the coverage of a message sent at time t is known by the sender at time $t + msgLatency + adaptNotif$ and it has just been explained that all entities that have received the message will be in some fail-safe mode $fSucc(m)$ by time $t + msgLatency + trans$. Since an entity cannot start acting until there is both a sufficient area of compatibility about it and it has knowledge of this, we deduce that a message only comes into effect at time $t_1 = t + msgLatency + max(trans, adaptNotif)$. The term $max(trans, adaptNotif)$ is the time corresponding to the waiting zone.

[1] We assume entities are not travelling close to light-speed and so relativistic effects are ignored.

Definition 5.1 (Mode Transition Time)**.** The mode transition time $tt(m,m')$ is the time it takes for an entity in mode m to reach the mode m'. In other words, if an entity initiates a mode switch from m to m', then once this amount of time has transpired, the mode state is ready to switch to m'. The actual switch however only happens when the entity signals to finalise the mode switch. ■

Definition 5.2 (The Fail-Safe Transition Function)**.** The fail-safe transition function $trans(m)$ yields the time it takes for an entity in mode m to reach a fail-safe mode, i.e. the time it takes for the mode state to time out, given that it has started such a transition. The definition is as follows:

$$trans(m) \stackrel{\text{def}}{=} setMax(\{tt(m,mF) | mF : \mathbb{FS}, m \rightsquigarrow mF\})$$

Appearing in the right hand side of this definition is a set of times built from the set of fail-safe successors to the mode m. These fail-safe successors, denoted by the dummy variable mF, are those modes which are both fail-safe and also successors to m as per the mode transition relation $\rightsquigarrow$. For each such successor mode, the set contains the transition time between m and the successor in question. The fail-safe transition function then returns the maximum value in this set. Intuitively, this is the worst-case time it will take for an entity to reach a fail-safe mode. ■

Definition 5.3 (The Fail-Safe Transition Constant)**.** The constant $trans$ is an upper bound on the time it takes for an entity to reach a fail-safe mode, given that it has initiated a mode transition. In other words, if an entity in any mode initiates a transition to any fail-safe mode $fSucc(m)$, then after some delay d less than or equal to $trans$, the mode state will have timed out and will be ready to enter $fSucc(m)$. The definition of $trans$ is as follows.

$$trans \stackrel{\text{def}}{=} setMax(\{trans(m) | m : \mathbb{M}\})$$

Here $trans(m)$ is the maximum time it takes for an entity in mode m to reach a fail-safe mode, and is defined in Definition 5.2. ■

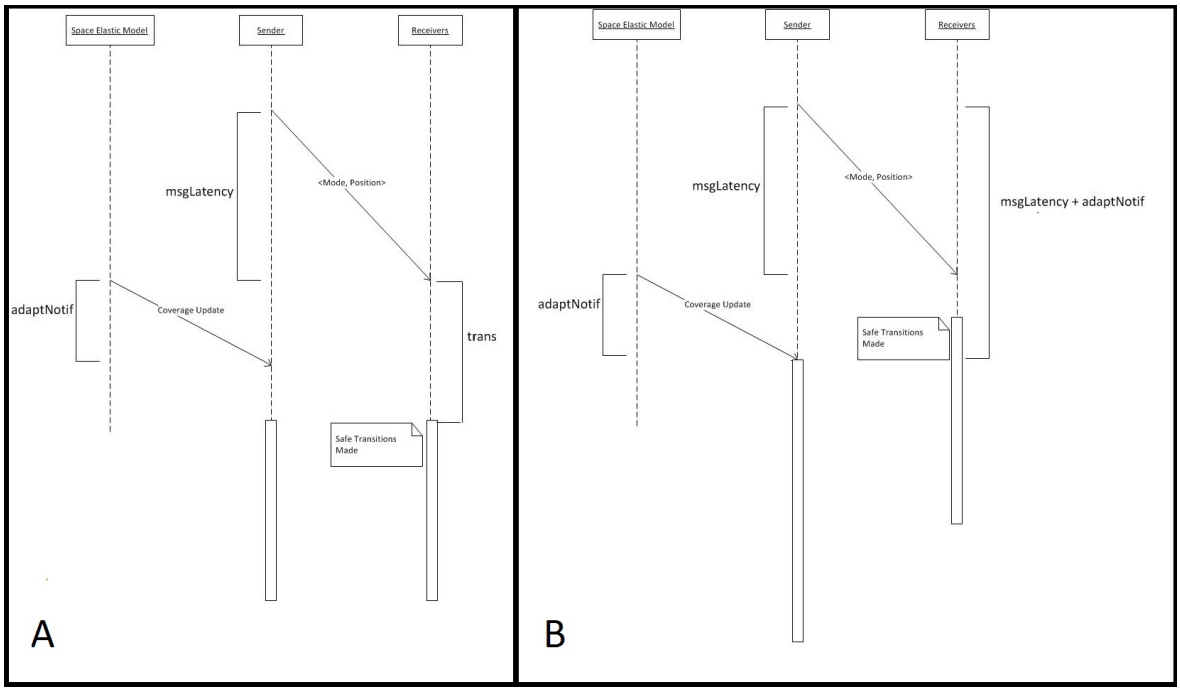


Figure 5.3: Case analysis of $trans \geq adaptNotif$. In (A) this is true and in (B) this is false. In each case, the larger time must have elapsed before the message comes into effect.

Recall from Table 3.1 that a Comhordú system includes for each of its modes m a constant $period(m)$, denoting the period of broadcast in that mode. An entity in mode m will broadcast mode m messages periodically

with this period. Hence, if the coverage is sufficient, the acting time provided by a message must be at least $period(m)$ i.e. the current message must guarantee safety until the next message comes into effect. Now, given that a message comes into effect at time t_1, we know that the next message will not come into effect until time $t_2 = t_1 + period(m)$, where m is the mode of the sender.

At time t_2, the sufficiency of the next message's coverage is evaluated. In the worst case, the coverage has degraded and the sender must immediately begin to transition to a fail-safe mode. Also, at any time, if the sender receives a message from another entity, and there is a possibility of incompatibility, then the sender will need to transition to a fail-safe mode. Hence, since it takes a time of $trans(m)$ for the sender to reach a fail-safe mode, the coverage must guarantee safety up until time $t_3 = t_2 + trans(m)$.

At time t_3, the region about the entity must be such that no entities are incompatible with **E**. Since the modes of surrounding entities are unknown to **E**, it must be assumed that the region at this time is at least the maximum of all the minimum distances of compatibility applied to m and any other mode m', to account for the worst-case scenario. In other words, the innermost region will be given by the function d_{max} of Definition 3.4.

Amalgamating all this reasoning, and assuming a worst case shrinking of the region about an entity based on $2 * s_{max}$, we get an expression for sufficient coverage. The predicate $suff(m,r)$ asserts whether the range r of a message sent by an entity in mode m is sufficient to allow this entity to act.

Definition 5.4 (Sufficient Coverage)**.** The predicate $suff$ defines when the coverage r of a message is sufficient given that the sender of the message was in mode m at the time of sending.

$$suff(m,r) \stackrel{\text{def}}{=} d_{max}(m) + 2 * s_{max} * [trans(m) + period(m) + max(trans, adaptNotif)] \leq r$$

Notice that in moving from left to right in this expression, the sub-terms correspond to the coverage regions going from the innermost region to the outermost region. ■

We have now defined what it means for coverage to be sufficient; we need only make a small deviation from this reasoning to specify when a received message can be ignored. It would be safe to assume that as long as the receiving entity **E** is outside the minimum sufficient coverage of the sending entity $\mathbf{E}'$, then it can ignore any messages sent by $\mathbf{E}'$. This is indeed the case. We must now ask, does **E** transition to a fail-safe mode if it is *inside* the coverage in question? This would be safe behaviour, but it would be overly conservative. This is because the mode of the receiver entity **E**, let's say m, is not taken into account in the calculation for the minimum sufficient coverage of the sender entity $\mathbf{E}'$, which only takes the sender's mode m' into account. This additional information of the receiver's mode allows a tighter bound on the region in which a receiver must react to messages. This new region is almost exactly the same as the minimum sufficient coverage of the sender with a slight alteration: the innermost region now becomes $minDistComp(m,m')$ instead of the overly conservative $d_{max}(m')$.

Definition 5.5 (The Possibly Incompatible Predicate)**.** The possibly incompatible predicate asserts whether a receiver entity in mode m' must begin a transition to a fail-safe mode given that it has just received a message from a sender entity in mode m which is at a distance r away. Recall the assumption earlier in this section that all entities transition to a fail-safe mode on receipt of a message from another entity. Well, the predicate $pInc$ allows for the relaxation of this assumption. This relaxation is possible because any given receiver will know it's own mode and can use this information to achieve a tighter bound than the sender assumes.

On the other hand, care must be taken by the receiver to correct for the actual distance of the sender on receipt of a message. Because messages received are assumed to have been sent $msgLatency$ time units ago, the actual position of the sender at the time of receipt may vary from the sent position by the amount $msgLatency * s_{max}$. Hence the distance r that a receiver passes to this predicate must be corrected by that factor; this is achieved by the listener process which is discussed in Section 5.3.3.

$$pInc(m,m',r) \stackrel{\text{def}}{=} r < minDistComp(m,m') + 2 * s_{max} * [trans(m) + period(m) + max(trans, adaptNotif)]$$

Notice the similarity between this definition and that of Definition 5.4. ■

5.3 Formal Protocol Specification

It is now time to use the previously discussed language of Section 4.1 to build a formal protocol process. The protocol process should exhibit a number of behaviours, as already discussed. Before launching straight into a description of the protocol process, let use briefly recap what these behaviours are: beaconing i.e. periodic broadcast of the current mode to surrounding entities; similar beaconing for alerting the environment of a mode switch; reaction to incoming messages; reaction to coverage degradation. With these in mind, let us move on to the formal protocol description.

The top-level protocol process **Protocol** is a parallel composition of three processes, namely **sleeping**, **dormant** and **listening**. The meaning of this process resides in the LTS whose states are all processes reachable from **Protocol**. We will refer to this as the top level LTS or the main LTS. It follows from the SOS of the software language that all states in this LTS will themselves be parallel compositions of three processes, each one a derivative of **sleeping**, **dormant** and **listening** respectively. The only way in which the parallel components can interact is by message passing. Hence it is useful to decompose the overall behaviour of the protocol into the behaviours of its individual parallel components and analyse these in isolation. This is exactly what will be done in the following.

$$\textbf{Protocol} \stackrel{\text{def}}{=} \textbf{sleeping} \mid \textbf{dormant} \mid \textbf{listening}$$

Remark 5.6 (Process Evolution vs. LTS State Transition)**.** Often instead of saying that a process **P** evolves into a process $\mathbf{P}'$, it is more intuitive to say that the LTS of **P** transitions from the state **P** to the state $\mathbf{P}'$. In fact sometimes, by a slight abuse of terminology, we will refer to an LTS as a "process", and say that it is in a certain state. This latter way of looking at things is more in line with our understanding of software, in which a program counter moves to various locations in a static piece of code. Often, the terms "state" and "process" will be used interchangeably in this context. ▲

Each of the processes **sleeping**, **dormant** and **listening** is given a definition. Each of these definitions mention other process names, which in turn have their own definitions. This gives rise to a mutually recursive set of definitions. It happens that these definitions can be divided into three disjoint groups corresponding to the three processes mentioned above. We refer to these sets of definitions as the broadcast, overlap and listening processes respectively. Notice that there is an abuse of terminology in referring to a set of definitions as a "process"; the reasons for this are similar to those given in the discussion of Remark 5.6. We also may refer to a set of definitions as a "component".

Due to the existence of parametrised processes and also due to the dense nature of the time domain chosen for this software language, most processes will give rise to infinite state LTSs. This raises the issue as to how to represent such objects graphically, if it is at all possible. The approach taken here is based on Timed Automata (TA). Equivalence classes of states, called locations, are drawn as circles. A location represents the set of all parameter instantiations of a parametrised process. Within a location, time parameters may change- this is continuous evolution of the system. All other changes correspond to discrete evolution of the system and are represented by edges. Of course, the actual semantics of this language are technically different from TA, but informally these pictures aid in understanding.

Prior to delving into the explanation of the protocol, let us take a high level view of how its sub-components are interconnected via channels, and briefly discuss the purpose of these connections. Figure 5.4 shows a high level schematic of the protocol with its three components. Not shown here are the channels connecting to external entity components; these can be seen in Figure 4.1. It can be seen in the diagram that the overlap component communicates with both the listener component and the broadcast component. There is no com-

munication between the listener and the broadcast components.

The channels *cTrans* and *cWake* allow the overlap component to signal to the broadcast component. A synchronisation across *cTrans* indicates that the overlap component is telling the broadcast component to go to sleep in accordance with a fail-safe transition that has occurred. A synchronisation on *cWake* comes from the overlap process and alerts the broadcast process to wake up because a new mode transition has just been completed.

The communication between the listener component and the overlap component is bidirectional. The channels *cBad* and *cAbort* allow the listener component to signal to the overlap component that action is necessary. In the case of *cBad*, this action is initiation of a transition to a fail-safe mode; in the case of *cAbort*, the necessary action is to abort the current mode transition. The channels *cPause* and *cUnpause* allow the overlap component to freeze/unfreeze the listener component while it performs some critical section activity.

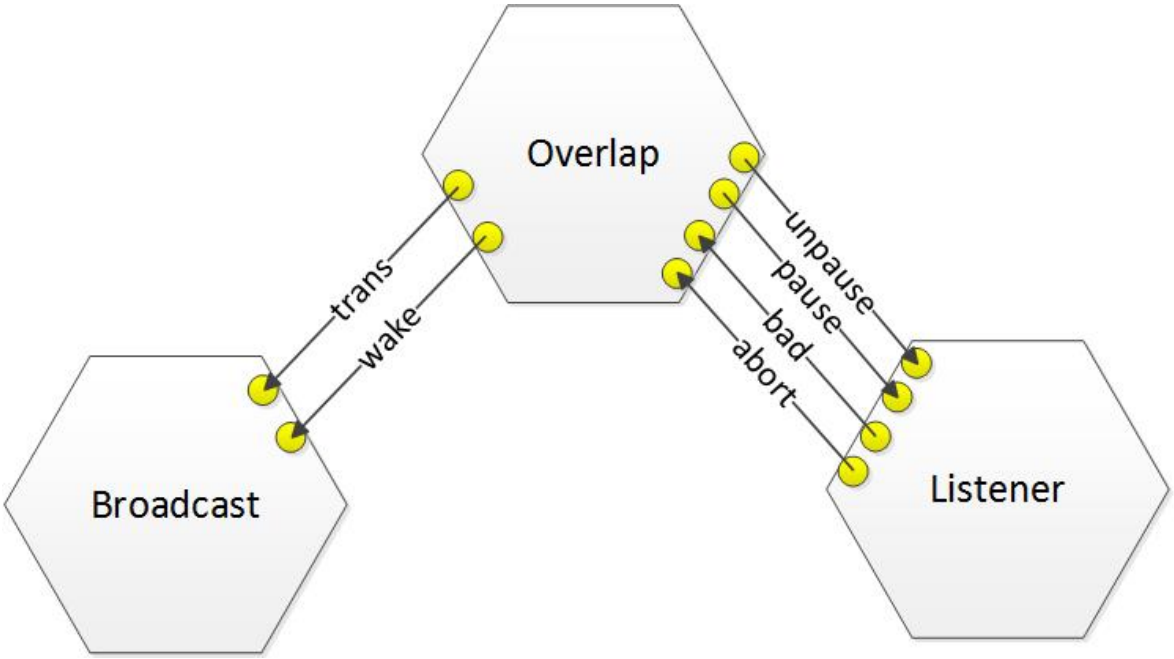


Figure 5.4: The three protocol components, connected via channels.

It is now time to explore this protocol process in more depth, first in terms of its individual sub-components and then as a whole. The following sections give the formal definition of the protocol via its three sub-components and also explore the various behaviours of these components. Section 5.3.1 presents the broadcast process i.e. the set of definitions relating to **sleeping**. In Section 5.3.2 the overlap process is covered i.e. the definitions corresponding to **dormant**. Section 5.3.3 gives definitions for the listener process i.e. those based around **listening**. Each of these sections also includes an exploration of the corresponding LTS by means of one or more traces. Finally, Section 5.3.4 contains traces of the overall protocol process i.e. the parallel composition of the three sub-components, showing the software component working as a whole.

5.3.1 Broadcast Process

Recall the example process given in Example 4.9. This was in fact a simplified version of the broadcast process that it presented here. Consequently, the following process definition should look familiar.

$$\mathbf{bcWait}(m,x) \stackrel{\text{def}}{=} cTrans?.\mathbf{sleeping} + \varepsilon(x).chanPos\langle l\rangle?.\mathbf{bcReady}(m,l) + cWake\langle m\rangle?.\mathbf{bcWait}(m,0)$$

$$\mathbf{sleeping} \stackrel{\text{def}}{=} cWake\langle m\rangle?.\mathbf{bcWait}(m,0) + cTrans?.\mathbf{sleeping}$$

$$\mathbf{bcReady}(m,l) \stackrel{\text{def}}{=} chanOutP\langle m,l\rangle!.\mathbf{bcWait}(m,period(m))$$

Remark 5.7 (Typing vs Naming Convention)**.** Here and in the remainder of the process descriptions to follow, often the types are omitted from variables in input prefixes. Instead, the symbol used is indicative of the type of the variable, as per the naming convention legend of Table 4.1. For example, when we write $chanPos\langle l\rangle?\mathbf{.}$, we are implicitly saying that l is a position because of the naming convention that has been adopted. The choice to adopt such a naming convention has been made since it results in the removal of much of the verbosity of explicit typing from terms. ▲

Figure 5.5 gives a graphical representation of the labelled transition system of the broadcast process. This process begins in the state **sleeping**. In this state, the corresponding entity, i.e. the entity within the context of which the protocol process resides, should[2] be in a fail-safe mode. As long as the entity remains in a fail-safe mode, the broadcast process should remain in the state **sleeping**. A change of mode may then occur, which will be effected by the overlap process, discussed in the next section, Section 5.3.2. On making a mode change, the overlap process should without delay inform this broadcast process of such a change, sending it a message on the channel *cWake* containing the appropriate mode m to which the entity has been switched. On receipt of such a message, the broadcast process will enter the **bcWait** state.

The purpose of the **bcWait** process is to periodically broadcast[3] messages to the environment containing the current mode and position. Notice from the definition of this process that it is a sum of three sub-processes, the middle of which is prefixed by a delay. This delay signifies the time remaining before the next message is broadcast. The process **bcWait** is entered via an input on the channel *cWake*, which synchronises with a matching output from the overlap component. The meaning of this is to "wake up" the process since a new mode transition has just been initiated. On entering this state, the time parameter is zero, and the process is immediately ready to begin a broadcast.

However, before doing so, it must first read the current position from the channel *chanPos*. After doing this, it enters the state **bcReady**, which is simply an intermediate process whose only job is to output the current mode and position to the environment through the interface along the channel *chanOutP*, before returning again to the wait state. Notice that this time on return to **bcWait** the time in the delay prefix guard is set to $period(m)$, i.e. the broadcast period of the mode m. The **bcWait** process must then delay by this amount of time before it can begin to broadcast another message; hence arises the periodicity of message broadcast.

We still have not covered the entirety of the behaviour of the **bcWait** process. Thus far, the other two components of the sum constituting its definition have been ignored. Let us now discuss these. The last component in the sum allows the process to constantly listen for mode updates along the channel *cWake* as it delays. This facilitates smooth transition from one mode to another. The first component in the sum, $cTrans?\mathbf{.sleeping}$, listens for an input on the channel *cTrans*. This channel is used by the overlap process to signal to the broadcast process that a fail-safe state has been entered. Hence, upon receipt of such a message, the broadcast process returns to **sleeping**.

Let us now examine a trace of the broadcast process. This trace will be similar to that of Example 4.9, and should capture the most important behaviours of the broadcast process. We begin in the state **sleeping** and allow some time to pass, let's say 13.6743 units of time.

$$\mathbf{sleeping} \xrightarrow{13.6743} \mathbf{sleeping}$$

Now, just to demonstrate the fact that two sequential delays are indeed possible, let us assume that another delay of, say, 15 time units occurs, again yielding no change in the state of the process

$$\mathbf{sleeping} \xrightarrow{15} \mathbf{sleeping}$$

[2]The word should here highlights the implicit assumption that the protocol is correct and that the entity in question has been started in an initial state, see Definition 6.1.

[3]The process in fact does not broadcast the message. Rather, it sends them to the interface component which buffers them and in turn, after sufficient time has passed, broadcasts them. However, this is cumbersome to say and we will henceforth refer, somewhat incorrectly, to outputs to the interface as broadcasts.

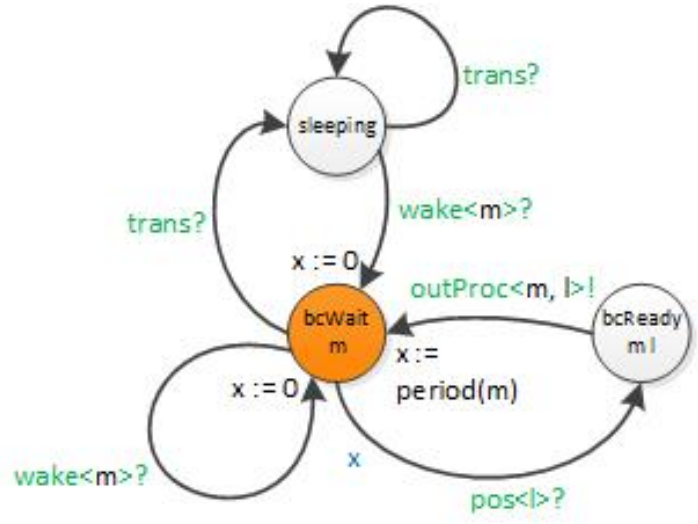


Figure 5.5: The broadcast process.

This could go on forever, but let us assume that it doesn't, and that eventually we reach a point at which an input is received along the channel *cWake* indicating a switch to some mode m.

$$\textbf{sleeping} \xrightarrow{cWake\langle m\rangle?} \textbf{bcWait}(m,0)$$

Now we are in an wait state, and the guard is timed out, so we are ready to read the current position and then perform a broadcast. For simplicity's sake, let's say that this is what happens. We will leave the position represented symbolically as l because there is no advantage in specifying an exact position and it would be cumbersome to write.

$$\textbf{bcWait}(m,0) \xrightarrow{chanPos\langle l\rangle} \textbf{bcReady}(m,l)$$

The interim state $\textbf{bcReady}(m,l)$ serves only to send out the broadcast message. Note that while a delay is possible here for the software process in isolation, it would not be so for the process in the context of an entity due to the violation of the predicate *noSyncEnt*. This predicate, discussed in Definition 4.23, ensures that a delay is only possible when synchronisation is not possible between any of the various sub-components of an entity, which in this case would not be so since the software process is ready to output to the interface, and interface is always willing to buffer messages. Therefore we assume time does not pass, and the broadcast happens immediately.

$$\textbf{bcReady}(m,l) \xrightarrow{chanOutP\langle m,l\rangle!} \textbf{bcWait}(m, period(m))$$

Now we are back in the wait state, except this time the delay guard is non-zero: it is set to the broadcast period $period(m)$ of the mode m. Let's assume then that some time d passes and the guard decreases somewhat. We assume that $d < period(m)$.

$$\textbf{bcWait}(m, period(m)) \xrightarrow{d} \textbf{bcWait}(m, period(m) - d)$$

Now let's say something interrupts the broadcast flow, a change of mode to the mode m'. This is signalled by a message from the overlap process along the channel *cWake*, just as the transition into the mode m from the initial fail-safe mode was signalled. This transition brings the broadcast process to a state in which the delay guard is zero and the mode is m'.

$$\textbf{bcWait}(m, period(m) - d) \xrightarrow{cWake\langle m'\rangle?} \textbf{bcWait}(m',0)$$

Notice the old mode m and the old remaining time $period(m) - d$ are reset instantaneously.

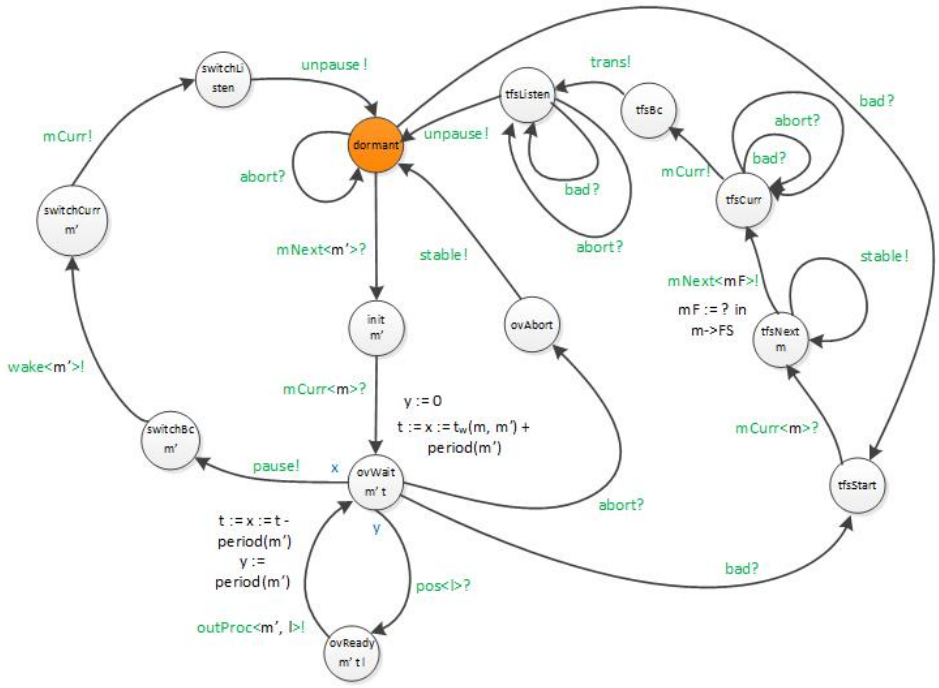


Figure 5.6: The overlap process.

We are now in a position in which the process can read the current position and then broadcast it, along with the mode m'. We have already seen this behaviour before with the mode m.

$$\textbf{bcWait}(m',0) \xrightarrow{chanPos\langle l\rangle} \textbf{bcReady}(m',l) \xrightarrow{chanOutP\langle m',l\rangle!} \textbf{bcWait}(m, period(m'))$$

Again we are back at the wait state and the delay guard is non-zero. A few things could happen here. A delay could occur and the guard would decrease. Alternatively, another mode change might happen via an input on the channel *cWake*, causing us to return to the timed out active state. However, we have already seen these behaviours so let us instead assume that an input occurs on the channel *cTrans*, which signals to the broadcast process that a transition to fail-safe mode has taken place. On receipt of this signal, the broadcast process re-enters the state **sleeping**. This transition is due to the first component in the sum which constitutes the **bcWait** process, i.e. $cTrans?$**,sleeping** .

$$\textbf{bcWait}(m, period(m')) \xrightarrow{cTrans?} \textbf{sleeping}$$

We are now back to where we started. This trace has hopefully demonstrated the chief behaviours of the broadcast process and shed light on its purpose in the overall context of the protocol.

5.3.2 Overlap Process

The overlap process facilitates smooth mode transitions, including transitions to fail-safe modes. The name "overlap" is chosen for this component because its periodic broadcasting behaviour overlaps that of the broadcast process while a mode switch is underway. Figure 5.6 graphically conveys the state-transition semantics of the overlap process. A brief explanation of this component is now given followed by some examples giving a more precise demonstration of its behaviour.

When the entity is in a stable state, the overlap process is in the state **dormant** and does nothing. When an entity decides to initiate a change of mode, which can occur spontaneously by the rule TEST given in

Table 4.15, the dormant process evolves to the waiting process **ovWait** via the intermediate process **init**.

From here, messages of the new mode are sent out to neighbouring entities, alerting them of the imminent mode and ensuring a compatible environment by the time a mode switch is due to take place. The waiting process **ovWait** is quite similar to the **bcWait** process discussed previously. The only difference is that it does not broadcast indefinitely. The waiting process with its parameters is written **ovWait**(m,t,t'). The first parameter is the pending mode, the second is the overall remaining wait time and the third is the time until the next broadcast. When enough time has passed, the waiting process moves into the state **switchBc**; from here a series of communications with other processes occur, finalising the mode switch. The exact value of the wait time, i.e. the time spent in **ovWait** depends on the current mode m and the pending mode m' and is defined in Definition 5.8.

Notice that in transitioning from **ovWait** to **switchBc**, an output is performed on the channel *cPause*. This forces a synchronisation with the listener process by pausing it. Such a synchronisation is necessary to ensure that the listener process does not have any pending messages to send to the overlap process, which may command it either to abort the current mode transition or begin a transition to a fail-safe mode.

The overlap component is now ready to finalise the mode switch. First, the broadcast process is alerted by **switchBc** of the mode switch via an output on the channel *cWake*. The overlap process then enters the state **switchCurr** and outputs on the channel *chanCurr*, causing the mode state to switch modes. Finally, the overlap process signals the end of its "critical section" activity to the listener process via an output on the channel *cUnpause* and re-enters the dormant state.

Definition 5.8 (Wait Time)**.** The wait time $t_w(m,m')$ is the time an entity must broadcast overlapping messages of the pending mode m' before finalising a mode switch from m to m'. It is defined as follows:

$$t_w(m,m') \stackrel{\text{def}}{=} max(tt(m,m'), msgLatency + max(adaptNotif, trans) + \Delta_t)$$

Intuitively, after messages of the pending mode have been broadcast for this amount of time, a number of conditions should hold:

- The mode state component should have timed out, granted that a mode change was indeed initialised before or at the time when broadcasting began. This means that the mode state is now ready to immediately accept a mode switch request i.e. a finalisation of the mode transition. This condition should hold due to the fact that the wait time is at least $tt(m,m')$, which is the time it takes for the mode state to time out when switching between modes m and m'.

- All potentially incompatible entities that received the initial broadcast message should now be in fail-safe modes. We know that other entities are running the protocol and hence they should react to messages by transitioning into a fail-safe mode if need be. It will take an entity a maximum time of $trans$ to transition to such a mode, and it will take the first message $msgLatency$ to deliver, hence after $t_w(m,m')$, which is at least as long as $msgLatency + trans + \Delta_t > msgLatency + trans$, all entities that received the first message should now be compatible.

  The term Δ_t is a small, strictly positive, delay that is added to ensure that all other entities have indeed transitioned to fail-safe modes. Without this it is possible, given the interleaving semantics, for the broadcasting entity to switch modes *before* all other entities have switched to fail-safe modes, albeit in the same time instant. This is simply a consequence of the language design: multiple events can happen at the same instant and the order is arbitrary.

- The sender entity knows the zone to which its first message has been delivered. This is due to the fact that the entity waits at least $adaptNotif + \Delta_t > adaptNotif$ time units after the delivery of the first message: once $adaptNotif$ time units have been exceeded, delivery notification is guaranteed to have been given by the SEM, so the entity will know the coverage by the time it is ready to switch modes. Notice the dual purpose of Δ_t, to ensure timed has elapsed both beyond $adaptNotif$ and beyond $trans$. Combined

with the previous fact, an entity can then know the area surrounding it in which all entities should be compatible with it. Based on this information it can then either decide to switch modes, if the area is sufficient, or abort the switch, if the area is insufficient.

■

Below is the formal definition of the overlap component.

$$\begin{aligned}
\textbf{dormant} &\stackrel{\text{def}}{=} \mathit{chanNext}\langle m'\rangle?.\textbf{init}(m') + \mathit{cAbort}?.\textbf{dormant} + \mathit{cBad}?.\textbf{tfsStart} \\
\textbf{init}(m') &\stackrel{\text{def}}{=} \mathit{chanCurr}\langle m\rangle?.\textbf{ovWait}(m', t_w(m,m') + \mathit{period}(m'), 0) \\
\textbf{ovWait}(m',t,t') &\stackrel{\text{def}}{=} \mathit{cAbort}?.\textbf{ovAbort} + \mathit{cBad}?.\textbf{tfsStart} + \\
&\qquad \varepsilon(t').\mathit{chanPos}\langle l\rangle?.\textbf{ovReady}(m',t,l) + \varepsilon(t).\mathit{cPause}!.\textbf{switchBc}(m') \\
\textbf{ovReady}(m',t,l) &\stackrel{\text{def}}{=} \mathit{chanOutP}\langle m',l\rangle!.\textbf{ovWait}(m', t - \mathit{period}(m'), \mathit{period}(m')) \\
\textbf{switchBc}(m') &\stackrel{\text{def}}{=} \mathit{cWake}\langle m'\rangle!.\textbf{switchCurr} \\
\textbf{switchCurr} &\stackrel{\text{def}}{=} \mathit{chanCurr}!.\textbf{switchListen} \\
\textbf{switchListen} &\stackrel{\text{def}}{=} \mathit{cUnpause}!.\textbf{dormant} \\
\textbf{ovAbort} &\stackrel{\text{def}}{=} \mathit{chanStable}!.\textbf{dormant} \\
\textbf{tfsStart} &\stackrel{\text{def}}{=} \mathit{chanCurr}\langle m\rangle?.\textbf{tfsNext}(m) \\
\textbf{tfsNext}(m) &\stackrel{\text{def}}{=} \mathit{chanNext}\langle \mathit{fSucc}(m)\rangle!.\textbf{tfsCurr} + \mathit{chanStable}!.\textbf{tfsNext}(m) \\
\textbf{tfsCurr} &\stackrel{\text{def}}{=} \mathit{cBad}?.\textbf{tfsCurr} + \mathit{cAbort}?.\textbf{tfsCurr} + \mathit{chanCurr}!.\textbf{tfsCurr} \\
\textbf{tfsCurr} &\stackrel{\text{def}}{=} \mathit{cTrans}!.\textbf{tfsListen} \\
\textbf{tfsListen} &\stackrel{\text{def}}{=} \mathit{cAbort}?.\textbf{tfsListen} + \mathit{cBad}?.\textbf{tfsListen} + \mathit{cUnpause}!.\textbf{dormant}
\end{aligned}$$

We have now seen the mode-switching behaviour of the overlap component. The remainder of its behaviours constitute reactions to events detected by the listener process. These reactions occur when the listener process signals to the overlap process that an exceptional event has occurred, by outputting on either the channel *cBad* or on *cAbort*. In the former case, a problem has arisen with the current mode, and it is necessary to immediately begin a transition into a fail-safe mode. In the latter case, the problem is with the pending mode, and it suffices to simply abort the current mode transition.

Aborting a transition is achieved by the process **ovAbort** via an output on *chanStable*. Note that in the dormant state, the overlap process ignores signals to abort via a self loop; there is no need to do anything if the state is already dormant, because in such a case, there is no pending mode transition to abort[4]. Also, the states **tfsStart** and **tfsListen** ignore signals to abort since a transition to a fail-safe mode is underway and so these signals must be outdated.

On receipt of an input on the channel *cBad*, the overlap process transitions to the state **tfsStart**; the overlap process has now begun a transition to a fail-safe mode. From **tfsStart**, the current mode *m* is read, transitioning to **tfsNext**. Then the existing mode transition is cancelled via a self-loop outputting on *chanStable*. The reason for a self-loop here is to retain the possibility of performing an abort should another mode change be initiated. This could happen due to the law $TEST$ which allows a stable entity to arbitrarily begin a legal mode transition. It is possible in this model, though not very realistic, that many such mode change initiations could happen within the same instant. Each time this happens, the mode switch is immediately aborted by the self loop. Eventually, it is reasonable to assume that these mode change initiations will cease, at which point the overlap process can initiate its own mode change into a fail-safe mode.

[4]This is not entirely correct. There *may* be a mode switch that has just been initiated this instant, but since no time has elapsed since its initiation, there is no need to abort, because the full wait time will still need to elapse before the mode switch is finalised.

The fail-safe mode is chosen according to an assumed function $fSucc$. Given a mode m, this function simply returns a fail-safe successor of that mode. Ideally, a non-deterministic choice of fail-safe successor would occur here, but the functional construct is easier to work with. The overlap process then waits in the state **tfsCurr** until the mode state times out, at which point it finalises the mode switch by an output on the channel *chanCurr*. The transition to a fail-safe mode is now complete. All that remains is to notify the other processes that this has occurred: the broadcast process is signalled on the channel $cTrans$; an output on $cUnpause$ synchronises with the listener process, ensuring that no signals on bad or abort remain.

We now study the overlap process in a slightly more formal manner with the use of traces. We will examine three traces in total, each trace corresponding to a different behaviour of the process. While these three traces do not cover every possible behaviour of this process, they have been chosen as they seem suitable to portray the main ways in which the process operates. The aim of presenting these traces is not to exhaustively search the state-space of the process, but to develop in the reader an understanding of how the process works. The behaviours we shall be considering are as follows:

- A "normal" mode switch with no interrupts from the listener process.
- An interrupt of a pending mode transition causing it to be aborted.
- A transition to a fail-safe mode.

Example 5.9 (A Normal Mode Switch with no Interrupts)**.** Let's begin with the trace of a normal mode switch with no interrupts. For the sake of this example, let's assume that the broadcast period for the mode m' is 9 and that the wait time between modes $t_w(m,m')$ is 21. Let us further assume that the current mode is m and that the transition time between the modes m and m', i.e. $tt(m,m')$ is 10. The overlap process starts in the state **dormant**. In this state, the process will idle indefinitely until a mode change is initiated. Let us say it idles for 56 time units.

$$\textbf{dormant} \xrightarrow{56} \textbf{dormant}$$

Eventually, we suppose that a mode change is in fact initiated, as per the rule $TEST$, and the pending mode becomes m'. Then the input upon the channel *chanNext* becomes enabled and along this channel the process reads the value of the next mode m' and enters the state **init**. Note that in the greater context of the entity time can't pass here because a synchronisation is available from the software component to the mode state component.

$$\textbf{dormant} \xrightarrow{chanNext\langle m'\rangle} \textbf{init}(m')$$

From this point, the process reads the current mode in order to calculate the waiting time $t_w(m,m')$, and then enters the waiting state **ovWait**. Note that the second parameter to the waiting process is 30, which is obtained by adding the wait time of 21 to the period of 9. The reason for adding the period to the wait time is as follows. Every time a broadcast is completed, the wait time remaining is decremented by the period $period(m')$ of the respective mode m'. This is so because broadcasts of m' are separated by exactly $period(m')$ units of time- hence every time a broadcast finishes, the wait time should decrease by $period(m')$, which is the amount of time that has elapsed since the last broadcast. The only exception to this is the initial broadcast, which happens instantaneously. Hence an additional $period(m')$ units of time is added to the wait time which will be immediately subtracted after the initial broadcast takes place.

$$\textbf{init}(m') \xrightarrow{chanCurr\langle m\rangle?} \textbf{ovWait}(m',30,0)$$

At this point, the last parameter to the waiting process is zero, and the transition to **ovReady** is possible via an input of the current position on the channel *chanPos*.

$$\textbf{ovWait}(m',30,0) \xrightarrow{chanPos\langle l\rangle?} \textbf{ovReady}(m',30,l)$$

This process serves only as an intermediate stage from which the message containing the mode and the position is broadcast.

$$\mathbf{ovReady}(m',30,l) \xrightarrow{chanOutP\langle m',l\rangle!} \mathbf{ovWait}(m',21,9)$$

Note that the wait time is now 21 and still no time has elapsed since the mode transition was initiated. This is in keeping with our intuitions. The process then idles until one of the terms in the choice construct times out.

$$\mathbf{ovWait}(m',21,9) \xrightarrow{9} cAbort?.\mathbf{ovAbort} + cBad?.\mathbf{tfsStart} + \boxed{\varepsilon(0).chanPos\langle l\rangle?.\mathbf{ovReady}(m',21,l)} + \varepsilon(12).cPause!.\mathbf{switchBc}(m')$$

Now, the third term in the sum has timed out. Thus it is again possible to read the position and then broadcast; let's say the position has now changed to l'. A box is used to highlight the sub-term responsible for the transition from its encompassing expression, see Remark 5.10 for more on this.

$$cAbort?.\mathbf{ovAbort} + cBad?.\mathbf{tfsStart} + \boxed{\varepsilon(0).chanPos\langle l\rangle?.\mathbf{ovReady}(m',21,l)} + \varepsilon(12).cPause!.\mathbf{switchBc}(m') \xrightarrow{chanPos\langle l'\rangle?} \mathbf{ovReady}(m',21,l')$$

From here, again the process outputs the mode and the position and moves back into the waiting state.

$$\mathbf{ovReady}(m',21,l') \xrightarrow{chanOutP\langle m',l'\rangle!} \mathbf{ovWait}(m',12,9)$$

On moving back to the waiting state, the time parameter is decreased from 21 to 12 to account for the 9 units of time that elapsed during the previous occupation of the waiting state. For this example, we assume that the waiting state again delays until it is ready to broadcast. Just to demonstrate the fact that delays may be split into two or more sub-delays, let us say that the delay of 9 is split into a delay of 7 followed by a delay of 2.

$$\begin{aligned}
&\mathbf{ovWait}(m',12,9)\\
&\xrightarrow{7} cAbort?.\mathbf{ovAbort} + cBad?.\mathbf{tfsStart} + \varepsilon(2).chanPos\langle l\rangle?.\mathbf{ovReady}(m',12,l)\\
&+ \varepsilon(5).cPause!.\mathbf{switchBc}(m')\\
&\xrightarrow{2} cAbort?.\mathbf{ovAbort} + cBad?.\mathbf{tfsStart} + \boxed{\varepsilon(0).chanPos\langle l\rangle?.\mathbf{ovReady}(m',12,l)}\\
&+ \varepsilon(3).cPause!.\mathbf{switchBc}(m')
\end{aligned}$$

Remark 5.10 (Verbosity of **ovWait** delay derivative)**.** The derivative of the process **ovWait** after a delay has occurred constitutes quite a verbose term. This is circumvented in the Coq model through the use of a generalisation of process definition bodies. Here however, we just write the entire term. To ease the burden on the reader, the sub-term of interest is enclosed in a box. This is the term that will contribute to the next transition. ▲

Again, we are in a position in which the third component in the sum has timed out and a broadcast can now be initiated by first reading the current position, which we now suppose is l'', and then outputting it along the output channel *chanOutP*.

$$cAbort?.\mathbf{ovAbort} + cBad?.\mathbf{tfsStart} + \boxed{\varepsilon(0).chanPos\langle l\rangle?.\mathbf{ovReady}(m',12,l)} + \varepsilon(3).cPause!.\mathbf{switchBc}(m') \xrightarrow{chanPos\langle l''\rangle?} \mathbf{ovReady}(m',12,l'') \xrightarrow{chanOutP\langle m',l''\rangle!} \mathbf{ovWait}(m',3,9)$$

We are now back in the waiting state **ovWait**, but this time the remaining wait time is less than the time until the next broadcast. This means that the transition to **switchBc** by an output on the channel *cPause* becomes available before the usual input on the channel *chanPos* does. For simplicity, let's assume a single delay of 3

time units occurs next.

$$\begin{aligned}\mathbf{ovWait}(m',3,9) \xrightarrow{3} & \mathit{cAbort}?.\mathbf{ovAbort} + \mathit{cBad}?.\mathbf{tfsStart} + \varepsilon(6).\mathit{chanPos}\langle l\rangle?.\mathbf{ovReady}(m',3,l) \\ & + \varepsilon(0).\mathit{cPause}!.\mathbf{switchBc}(m')\end{aligned}$$

We assume that the output on *cPause* happens, i.e. that the listener process is in a stable state and can synchronise on this channel. The necessity of synchronising on the pause channel before a mode switch takes place is evident when one considers the fact that the listener process relies on reading the next and current modes in order to assess whether to send a message on *cAbort* or *cBad*. Without such a synchronisation, certain permutations involving the listener process reading the current/pending modes and the overlap process writing them lead to undesirable situations. For example, consider the situation in which the listener process signals on *cAbort* instead of *cBad* due to the fact that a mode switch takes place immediately after it reads the pending mode. In this case, the problem is now in fact with the new current mode, i.e. the old pending mode, but the information the listener process has is out of date, and it sends a message on *cAbort* instead of on *cBad*. This is a standard "critical section" problem, common in interleaving models of concurrency.

$$\begin{aligned}\mathit{cAbort}?.\mathbf{ovAbort} + \mathit{cBad}?.\mathbf{tfsStart} + \varepsilon(6).\mathit{chanPos}\langle l\rangle?.\mathbf{ovReady}(m',3,l) \\ + \boxed{\varepsilon(0).\mathit{cPause}!.\mathbf{switchBc}(m')} \xrightarrow{\mathit{cPause}!} \mathbf{switchBc}(m')\end{aligned}$$

From here, the mode state is updated with the change of mode by outputting on the channel *chanCurr*, after which the broadcast process is notified of the mode switch by an output on the channel *cWake*.

$$\mathbf{switchBc}(m') \xrightarrow{\mathit{cWake}\langle m'\rangle!} \mathbf{switchCurr} \xrightarrow{\mathit{chanCurr}!} \mathbf{switchListen}$$

All that remains is to unpause the listener process before returning to the dormant state once again.

$$\mathbf{switchListen} \xrightarrow{\mathit{cUnpause}!} \mathbf{dormant}$$

▲

Example 5.11 (Aborting a Mode Transition)**.** Here we explore a scenario in which a mode transition is initiated and then aborted before it can be completed. The decision to abort the mode transition comes from the listener process. This decision occurs when a critical event is discovered relative to the pending mode. Such an event is either a degradation of coverage or an incompatible message from another entity.

Let us begin by considering a corner case. It is assumed here that we have an entity in mode m and that the listener process decides to signal to the overlap process to abort the current mode transition when the overlap process is still in the dormant state. This exceptional case may arise if a new mode transition m' has just been initiated this instant by the rule TEST and at the same time, a message arrives from another entity which is incompatible with the mode m'. Now, since no time has been spent waiting on this mode transition to occur, and hence to finalise this transition would still require waiting the full wait time $t_w(m,m')$, there is no need to abort the mode transition. Instead, the abort message is essentially ignored.

$$\mathbf{dormant} \xrightarrow{\mathit{cAbort}?} \mathbf{dormant}$$

After this, we assume the events previously discussed occur: the overlap process reads the pending mode and then the current mode, enters the waiting state **ovWait**, and then immediately performs an input of the current position followed by an output on *chanOutP*. We assume in this scenario that the wait time $t_w(m,m')$ is

13 and the broadcast period $period(m')$ of the mode m' is 8.

$$\mathbf{dormant} \xrightarrow{chanNext\langle m'\rangle} \mathbf{init}(m') \xrightarrow{chanCurr\langle m\rangle?} \mathbf{ovWait}(m', 21, 0)$$
$$\mathbf{ovWait}(m', 21, 0) \xrightarrow{chanPos\langle l\rangle?} \mathbf{ovReady}(m', 21, l) \xrightarrow{chanOutP\langle m', l\rangle!} \mathbf{ovWait}(m', 13, 8)$$

After this first broadcast message has been sent out, the process may now delay in the waiting state **ovWait**. We assume that it delays for 5 time units.

$$\mathbf{ovWait}(m', 13, 8) \xrightarrow{5} cAbort?.\mathbf{ovAbort} + cBad?.\mathbf{tfsStart} + \varepsilon(3).chanPos\langle l\rangle?.\mathbf{ovReady}(m', 13, l) + \varepsilon(8).cPause!.\mathbf{switchBc}(m')$$

Now, in normal circumstances, this process would continue as in the previous example, delaying until the guard on the third term in the sum timed out, at which point an input of the current position and a broadcast would follow. However, we assume what happens next is that the operation of the overlap process is interrupted by a message on the channel *cAbort* from the listener process, signalling to abort the current mode transition. The reaction to such a message is to transition to the state **ovAbort**.

$$\boxed{cAbort?.\mathbf{ovAbort}} + cBad?.\mathbf{tfsStart} + \varepsilon(3).chanPos\langle l\rangle?.\mathbf{ovReady}(m', 13, l) + \varepsilon(8).cPause!.\mathbf{switchBc}(m') \xrightarrow{cAbort?} \mathbf{ovAbort}$$

From here, the existing mode transition is aborted by an output on the channel *chanStable*, and the overlap process returns to the dormant state, awaiting the initiation of another mode transition.

$$\mathbf{ovAbort} \xrightarrow{chanStable!} \mathbf{dormant}$$

▲

Example 5.12 (A Fail-Safe Transition)**.** In this example, we will examine what happens when the listener process outputs on the channel *cBad*, alerting the overlap process to perform a transition to a fail-safe mode. In either of the states **dormant** or **ovWait**, the overlap process may input on the channel *cBad* and then evolve to the state **tfsStart**. In other words, the following transitions are possible:

$$\mathbf{dormant} \xrightarrow{cBad?} \mathbf{tfsStart}$$

and

$$\mathbf{ovWait}(m, t, t') \xrightarrow{cBad?} \mathbf{tfsStart}$$

In either case, the derivative process is **tfsStart**. From here, a fail-safe transition is initiated and eventually completed. First, the current mode is read and the process transitions to the state **tfsNext**.

$$\mathbf{tfsStart} \xrightarrow{chanCurr\langle m\rangle?} \mathbf{tfsNext}(m)$$

Let's say there is a mode switch pending: a self-looping edge from **tfsNext** outputting on *chanStable* allows this to be canceled.

$$\mathbf{tfsNext}(m) \xrightarrow{chanStable!} \mathbf{tfsNext}(m)$$

This causes the mode state to return to a stable state m. From here, it is now possible for the overlap process to begin a transition to some fail-safe mode $fSucc(m)$ by writing it out on the channel *chanNext*, causing the mode state to enter a transitional state.

However, it is also possible that in the meantime, a different mode transition to a mode m' is initiated by the rule $TEST$. Let us say that this happens. Now it is no longer immediately possible for the overlap process

to write the fail-safe mode to the mode state. This is why the state **tfsNext** has a self looping edge allowing it to abort an arbitrary number of such mode transitions that are initiated by the rule $TEST$. The self looping transition is taken, yet again freeing up the mode state for the initiation of a transition to a fail-safe mode. Note that the accompanying mode state, not depicted here, is assumed to be in the process of a mode transition every time a transition on *chanStable* occurs.

$$\mathbf{tfsNext}(m) \xrightarrow{chanStable!} \mathbf{tfsNext}(m)$$

Of course, it is possible that such spurious mode transition initiations could go on indefinitely. However, we assume that they don't, and that the overlap process eventually gets to write out the mode $fSucc(m)$ on the channel *chanNext*, thus initiating in the mode state a transition to the fail-safe mode $fSucc(m)$.

From here, a mode is chosen according to an assumed, uninterpreted function $fSucc$. The chosen mode is then output on the channel *chanNext* to initiate a transition to that mode, and the overlap process moves into the state **tfsCurr**.

$$\mathbf{tfsNext}(m) \xrightarrow{chanNext\langle fSucc(m)\rangle!} \mathbf{tfsCurr}$$

From here, the overlap process simply idles until the transition in the mode state has timed out. In the meantime, it is possible that the listener process may send messages on *cBad* or *cAbort*; these will be ignored. Let's say that the mode transition time $tt(m, fSucc(m))$ from m to $fSucc(m)$ is 9 time units and the overlap process initially delays by 4 time units, leaving 5 time units of a delay left until the mode state times out.

$$\mathbf{tfsCurr} \xrightarrow{4} \mathbf{tfsCurr}$$

At this stage, it is possible that the listener process may send a message along the channel *cBad*; this may be ignored because a transition to a fail-safe mode is already underway.

$$\mathbf{tfsCurr} \xrightarrow{cBad?} \mathbf{tfsCurr}$$

Now we assume that a further 3.5 time units of a delay elapses, leaving 1.5 time units to delay until the mode state times out.

$$\mathbf{tfsCurr} \xrightarrow{3.5} \mathbf{tfsCurr}$$

Then let's suppose the listener process sends another message, this time an output on the channel *cAbort*. This will be ignored just as was the previous message from the listener process.

$$\mathbf{tfsCurr} \xrightarrow{cAbort?} \mathbf{tfsCurr}$$

The remaining 1.5 time units elapse and the mode state times out. The timing out of the mode state is not shown here as it is outside the scope of the overlap process.

$$\mathbf{tfsCurr} \xrightarrow{1.5} \mathbf{tfsCurr}$$

Now, the mode state has timed out, and it is possible for the overlap process to complete the mode transition to $fSucc(m)$ by outputting on the channel *chanCurr*.

$$\mathbf{tfsCurr} \xrightarrow{chanCurr!} \mathbf{tfsBc}$$

It is now necessary to notify the broadcast process that a fail-safe mode has been entered. This is done by means of an output on the channel *cTrans*.

$$\mathbf{tfsBc} \xrightarrow{cTrans!} \mathbf{tfsListen}$$

Now, it is possible that the listener component is currently processing some event. If this is the case, it is desirable to wait until said processing is finished before returning to the dormant state. This is to ensure that another fail-safe transition isn't initiated needlessly based on out-of-date information. Let's say that the listener is indeed computing, and outputs on the channel *cBad*. Then a self loop inputting on the same channel allows this to happen without consequence.

$$\textbf{tfsListen} \xrightarrow{cBad?} \textbf{tfsListen}$$

Finally then, an output on the channel *cUnpause* synchronises with the listener process, returning to the dormant state; the synchronisation ensures that the listener is no longer in the process of some reaction to an event.

$$\textbf{tfsListen} \xrightarrow{cUnpause!} \textbf{dormant}$$

▲

Let us now summarise this exposition of the overlap process with a look at an augmented version of its state transition diagram which summarises the process behaviour at a high level. Figure 5.7 is similar to Figure 5.6, the only difference being that states are grouped together based on their functions. Groups of states in the figure are labelled and when there is more than one state in a group it is highlighted with a translucent ellipse. There are six groups of states into which this process is logically sub-divided:

- **dormant**: In this state, the process waits until a mode switch is started.
- **init**: In this state, the process is preparing to begin broadcasting.
- Broadcasting (**ovWait** and **ovReady**): This group is located at the bottom of the diagram, highlighted by a shaded region in the shape of a tennis ball. In these states, the process periodically broadcasts to the environment, much like the broadcast process does.
- Switching modes (**switchBc**, **switchCurr** and **switchListen**): This group of states constitutes the egg-shaped region to the left of the diagram. The states form a sequence over which a mode switch is being finalised via a number of synchronisations with other components.
- Aborting a transition (**ovAbort**).
- Transitioning to a fail-safe mode (**tfsStart**, **tfsNext**, **tfsCurr** and **tfsBc**). Notice the shaded region to the right of the diagram that looks somewhat like a daisy petal. This constitutes the group of states which characterise the behaviour of an entity as it switches to a fail-safe mode.

5.3.3 Listener Process

Below is the set of process definitions corresponding to the listener process. A graphical representation of these is given in Figure 5.8. These processes are briefly explained here and then a number of traces are given demonstrating their behaviour according to the semantics of the software language.

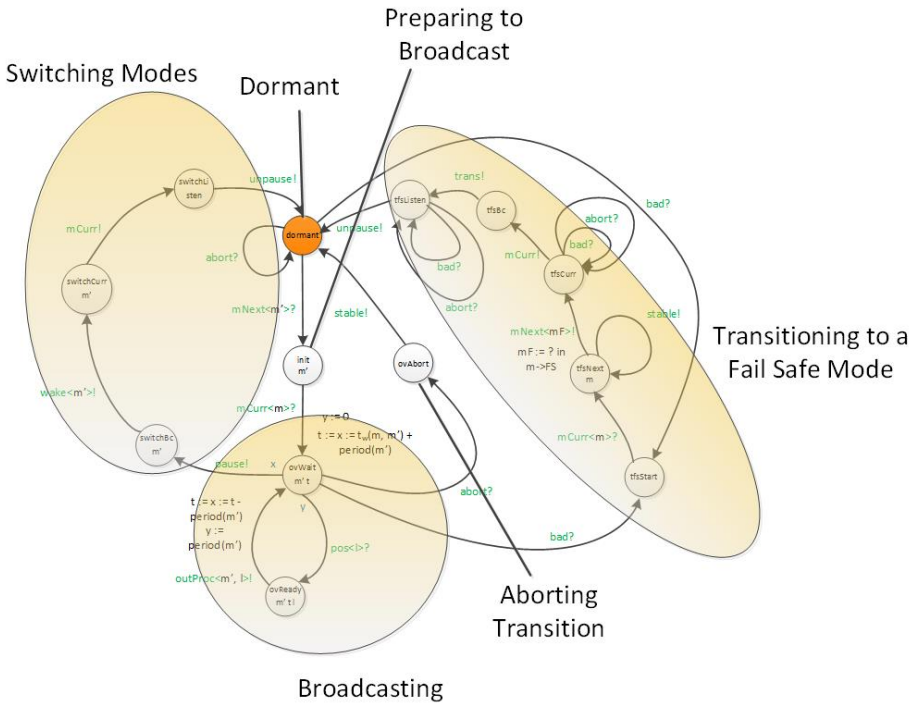


Figure 5.7: The overlap process, with states classified into operational sections.

$$\begin{aligned}
\textbf{listening} &\stackrel{\text{def}}{=} chanAN\langle r, m''\rangle?.\textbf{checkRange}(m'', r) + chanInP\langle m'', l'\rangle?.\textbf{gotMsg}(m'', l') \\
&\quad + cPause?.\textbf{paused} + cUnpause?.\textbf{listening} \\
\textbf{checkRange}(m'', r) &\stackrel{\text{def}}{=} \textbf{if } suff(m'', r) \textbf{ then listening} + \textbf{if } \neg suff(m'', r) \textbf{ then rangeBad}(m'') \\
\textbf{rangeBad}(m'') &\stackrel{\text{def}}{=} chanCurr\langle m\rangle?.\textbf{currEq}(m, m'') \\
\textbf{currEq}(m, m'') &\stackrel{\text{def}}{=} \textbf{if } m = m'' \textbf{ then badOvlp} + \textbf{if } m \neq m'' \textbf{ then currOK}(m'') \\
\textbf{badOvlp} &\stackrel{\text{def}}{=} cBad!.\textbf{listening} \\
\textbf{currOK}(m'') &\stackrel{\text{def}}{=} chanStable?.\textbf{listening} + chanNext\langle m'\rangle?.\textbf{nextEq}(m', m'') \\
\textbf{nextEq}(m', m'') &\stackrel{\text{def}}{=} \textbf{if } m' = m'' \textbf{ then abortOvlp} + \textbf{if } m' \neq m'' \textbf{ then listening} \\
\textbf{abortOvlp} &\stackrel{\text{def}}{=} cAbort!.\textbf{listening} \\
\textbf{gotMsg}(m'', l') &\stackrel{\text{def}}{=} chanPos\langle l\rangle?.\textbf{gotRange}(m'', |l - l'| - s_{max}.msgLatency) \\
\textbf{gotRange}(m'', r) &\stackrel{\text{def}}{=} chanCurr\langle m\rangle?.\textbf{currPincCheck}(m, m'', r) \\
\textbf{currPincCheck}(m, m'', r) &\stackrel{\text{def}}{=} \textbf{if } \neg pInc(m, m'', r) \textbf{ then currComp}(m'', r) + \textbf{if } pInc(m, m'', r) \textbf{ then badOvlp} \\
\textbf{currComp}(m'', r) &\stackrel{\text{def}}{=} chanStable?.\textbf{listening} + chanNext\langle m'\rangle?.\textbf{nextPincCheck}(m', m'', r) \\
\textbf{nextPincCheck}(m', m'', r) &\stackrel{\text{def}}{=} \textbf{if } \neg pInc(m', m'', r) \textbf{ then listening} + \textbf{if } pInc(m', m'', r) \textbf{ then abortOvlp} \\
\textbf{paused} &\stackrel{\text{def}}{=} cUnpause?.\textbf{listening}
\end{aligned}$$

The "central" process in all these definitions is **listening**. This process listens for both incoming adaptation notification messages and incoming broadcasts from other entities. On receipt of such a message, the process evolves into either **checkRange** for handling a coverage adaptation notification or **gotMsg** for handling a broadcast. In either case, a computation is performed on the message to assert whether or not it is incompatible

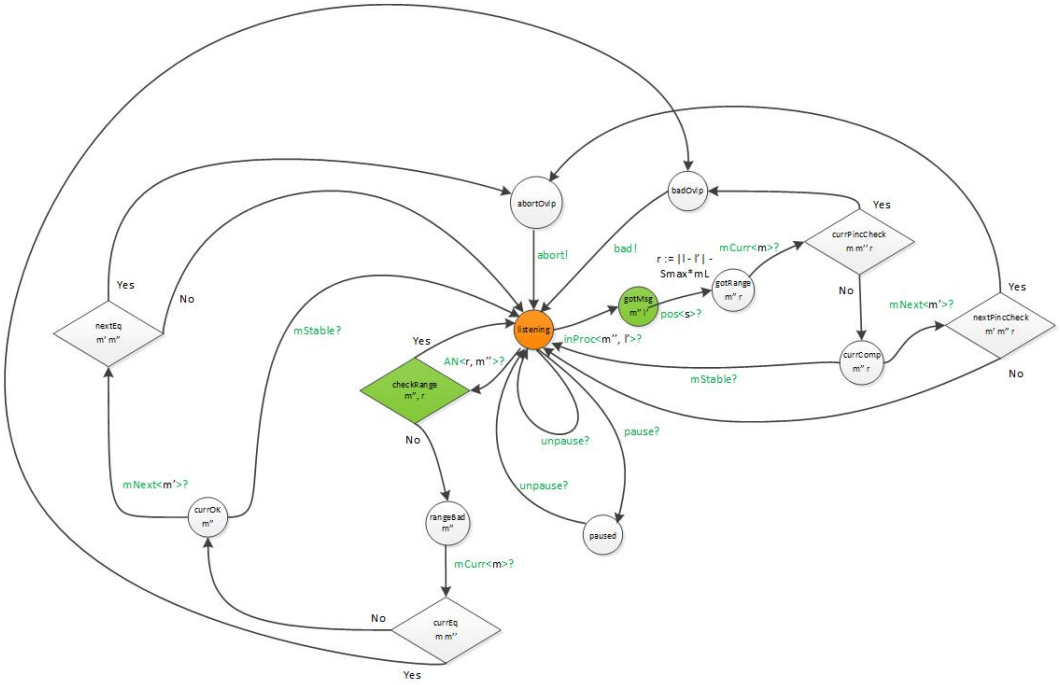


Figure 5.8: The listener process.

with the current or the next mode. For an incoming message, incompatibility is tested using the predicate *pInc*, defined in Definition 5.5: the expression $pInc(m, m', r)$ asserts whether a distance of r is sufficient to separate two entities in modes m and m' respectively. For a notification message, the predicate $suff$, defined in Definition 5.4, tests whether or not the delivery range of a message is sufficient to ensure safe operation in the corresponding mode.

When an incoming message is deemed unsatisfactory, i.e. either a range of delivery was too small or another entity is too close based on its mode, then the overlap process is signalled with an output on one of the channels *cBad* or *cAbort*. Communication on the former channel signals to the overlap process that it should begin a transition to fail-safe mode. The latter is used to indicate to the overlap process that it should abort the transition to the next mode.

Let us now roughly explain the processes involved in handling a coverage notification. The process **rangeBad** is reached when **checkRange** computes that the range of delivery for the received notification was too small. From here, the listener evolves to **currEq** and checks first whether the insufficiency was with the current mode. If so a signal must be sent to the overlap component on the channel *cBad*. This is done by the state **badOvlp**. Otherwise, the process evolves to **currOK**. From here, it is possible that there is no next mode, which is tested by an input on *chanStable* from the mode state. In this case, the listener returns to its listening state. Otherwise, the next mode is compared to the received mode by the process **nextEq**. If they are equal, then a message must be sent on *cAbort* to the overlap component. This is done by **abortOvlp**. Otherwise the process effectively returns to the listening state which is essentially the "else" branch of the process **nextEq**.

The processes involved in handling an incoming message are similar to those for handling a coverage notification. From the process **gotMsg**, the current position is read and the listener evolves to **gotRange**. The range calculated is the difference in the received position and the current position of the entity, minus a worst-case correction factor of $s_{max} * msgLatency$ to account for the fact that the sender may have moved since sending. This range is now checked against the current mode by the process **currPincCheck** to see if there is an incompatibility there. If so, **badOvlp** is reached and a message on the channel *cBad* signals a transition to a fail-safe mode. Otherwise, the current mode is compatible and the state **currComp** is reached. From here, either the mode state is stable and the listener returns to the listening state, or there is some next mode, which is checked for compatibility against the received message by the state **nextPincCheck**. An incompatibility

here warrants a signal on *cAbort* by the process **abortOvlp**. Otherwise the process effectively returns to the listening state.

The state **paused** exists so that the overlap component can freeze the listener in place while it effects a mode switch. This is necessary so that the listener process does not signal to the overlap process based on out-of-date information. For example, let's say an incompatibility with the next mode is found by the listener process and it enters the state **abortOvlp**, ready to signal to the overlap process on the channel *cAbort*. But then imagine the overlap process in the meantime switches modes, so that the next mode now becomes the current mode. Then the listener component should send a message on *cBad*, rather than *cAbort*, because the incompatibility will now be with the current mode. By adding the paused state to the listener component *and* by ensuring that the overlap component always pauses before a mode switch, this unsatisfactory situation is avoided.

Let us now examine the behaviour of the listener process by means of two traces. The first trace will cover the reaction of the listener process to an adaptation notification message, while the second will portray the behaviour of the listener process on receiving a broadcast message from another entity. These traces will differ slightly from those seen before; instead of following one path of execution, a number of options will be considered for the continuation of the trace. These variations could of course all be done as separate traces, but it is much more concise to present them as alternate branches of a single trace since they all share common prefixes. Let us now present these two traces.

Example 5.13 (Reaction of Listener Process to Adaptation Notification Message)**.** The listener process begins in the state **listening**. From here, we suppose that it receives a message on the channel *chanAN* containing a mode m'' and a delivery radius r. The meaning of this message is that a broadcast message with mode m'' has been delivered to a radius r at a time $adaptNotif$ units of time previous to the time the adaptation notification message is received.

$$\textbf{listening} \xrightarrow{chanAN\langle m'',r\rangle?} \textbf{checkRange}(m'',r)$$

The first thing the listener process does, is to check whether the range r is sufficient for the mode m'', using the predicate $suff$ which is defined in Definition 5.4. This can be seen in the body of the definition of **checkRange**. If the predicate turns out to be true, then the listener process simply behaves the same as the **listening** state, thereby ignoring the message. We assume in this case that the predicate evaluates to false i.e. that coverage was insufficient. Then the guard on the second conditional branch in the body of **checkRange** becomes true and the process behaves as **rangeBad**. Hence an input on the channel *chanCurr* is possible, bringing the process to the state **currEq**.

$$\textbf{checkRange}(m'',r) \xrightarrow{chanCurr\langle m\rangle?} \textbf{currEq}(m,m'')$$

In this state, the process inspects whether the current mode is equal to the mode received in the adaptation notification message. Suppose this is the case; then the process evolves like **badOvlp** and returns to the initial state. Since the problem is with the current state, a signal is sent on the channel *cBad*, notifying the overlap component to initiate a fail-safe transition.

$$\textbf{currEq}(m,m'') \xrightarrow{cBad!} \textbf{listening}$$

It is also possible that the mode m'' differs from the current mode m. In this case, the listener process behaves like **currOK**. From here, there are two cases to consider. Either the mode state is stable, or there is a pending mode transition to a new mode m'. In the former case, the mode state will be willing to output on the channel *chanStable*, and the listener process will synchronise with this by inputting on the same channel and returning to the initial state.

$$\textbf{currEq}(m,m'') \xrightarrow{chanStable?} \textbf{listening}$$

Alternatively, there is a pending mode transition, and the mode state is willing to output the next mode, call it m', on the channel *chanNext*. In this case, the listener process inputs said mode and evolves to the state

nextEq.

$$\textbf{currEq}(m, m'') \xrightarrow{chanNext\langle m'\rangle?} \textbf{nextEq}(m', m'')$$

A check is again performed here this time to test whether or not the received mode m'' is equal to the pending mode m'. If the two are equal, then the process sends an abort message to the overlap process and then returns to the initial state.

$$\textbf{nextEq}(m', m'') \xrightarrow{cAbort!} \textbf{listening}$$

If the received mode is not equal to the pending mode, then the process simply continues on as if it were **listening**. ▲

Example 5.14 (Reaction of Listener Process to Broadcast Message)**.** Let us now give a trace demonstrating the reaction of the listener process to a broadcast message that is received from another entity. As in the last example, the listener process begins in the state **listening**. From here, it inputs the broadcast message, which comes from the interface component of the entity since it gets buffered there after coming from the other entity. The state reached is **gotMsg**. The values m'' and l' constitute the broadcast message that is input and bind to parameters of the derivative process.

$$\textbf{listening} \xrightarrow{chanInP\langle m'', l'\rangle?} \textbf{gotMsg}(m'', l')$$

From here, the listener process reads the current position l, i.e. that of the entity in which it is running, and then evolves to the state **gotRange**. The parameter r of this new state is calculated as the distance $|l - l'| - s_{max} * trans$ between the current position l and the position l' that was sent in the broadcast message. Notice that we subtract the term $s_{max} * trans$ to account for the fact that the entity may have moved since sending the message. Subtracting accounts for the worst possible case i.e. the sender entity could be no closer.

$$\textbf{gotMsg}(m'', l') \xrightarrow{chanPos\langle l\rangle?} \textbf{gotRange}(m'', |l - l'|)$$

Now, we first check whether or not the sender's mode m'' and distance r are possibly incompatible with the current mode m, as per the predicate *pInc*. This is done by reading the current mode on the channel *chanCurr* and then evolving to the state **currPincCheck**, the body of which uses the predicate *pInc* to decide what action to take next.

$$\textbf{gotRange}(m'', r) \xrightarrow{chanCurr\langle m\rangle?} \textbf{currPincCheck}(m, m'', r)$$

Let's first consider the case in which the message is possibly incompatible with our current mode. In this case, *pInc* evaluates to true and the process behaves just like **badOvlp**: it sends a message on channel *cBad* and returns to the original state.

$$\textbf{currPincCheck}(m, m'', r) \xrightarrow{cBad!} \textbf{listening}$$

If the predicate turns out to be false, then the process continues behaving like **currComp**. Now there are two cases to consider. If the mode state is stable, then it can output on the channel *chanStable* and the listener process will synchronise with this by inputting on said channel and evolving to the state **listening**.

$$\textbf{currPincCheck}(m, m'', r) \xrightarrow{chanStable?} \textbf{listening}$$

On the other hand, if there is a pending mode, it is read by the listener process, which then evolves to **nextPincCheck**.

$$\textbf{currPincCheck}(m, m'', r) \xrightarrow{chanNext\langle m'\rangle} \textbf{nextPincCheck}(m', m'', r)$$

Now, if the received mode m and the distance r are incompatible with the pending mode m', the listener process will behave as **abortOvlp**, and will send an abort to the overlap process telling it to abort the current mode

transition, before evolving to the original state **listening**.

$$\textbf{nextPincCheck}(m', m'', r) \xrightarrow{cAbort!} \textbf{listening}$$

Alternatively, if the received mode and distance are compatible with the pending mode, the process simply continues as if it were in the original state **listening**. ▲

We have now seen the two main behaviours of the listener process: reaction to an adaptation notification message and reaction to a broadcast message from another entity. In summary, these behaviours are similar in that they consist of 3 sub-cases, which we will refer to as bad, abort and neutral. In the bad case, the necessary reaction is to transition to a fail-safe mode. In the abort case, the pending mode transition must be aborted. For the neutral case, no action is necessary and the listener component simply returns to the state **listening**.

Now we also must not forget that this listener process may be paused by the overlap process, so that the overlap process may perform the critical task of switching modes. Though this behaviour is quite uninteresting, we will explicitly show it here just for completeness.

$$\textbf{listening} \xrightarrow{cPause?} \textbf{paused}$$

When the overlap process is finished in its critical section, it will send a message on the channel $cUnpause$, allowing the listener to return to its original state.

$$\textbf{paused} \xrightarrow{cUnpause!} \textbf{listening}$$

Before moving on, let us recap on what the listener process does. Figure 5.9 breaks the behaviour of the listener process into five major sections. These are:

- **listening**: Listening for incoming broadcasts and coverage notification messages.
- **paused**: The listener is frozen while the overlap process finishes some critical section computation.
- Handling a coverage notification (**checkRange**, **rangeBad**, **currEq**, **currOK** and **nextEq**).
- Handling a broadcast from another entity (**gotMsg**, **gotRange**, **currPincCheck**, **currComp** and **nextPincCheck**).
- Notifying overlap process of a critical event (**badOvlp** and **abortOvlp**): When one of these states is reached a critical event has occurred, i.e. a coverage degradation has been detected or a broadcast from another potentially incompatible entity has been received. The critical even can be relative to either the current mode, in which case the state reached will be **badOvlp**, or to the pending mode, in which case the state reached will be **abortOvlp**.

5.3.4 Behaviour of the Overall Process

In this section, traces of the full protocol process are explored. These traces give an indication of how the sub-components of the protocol interact with each other, as well as with "external" components i.e. non-software components of the entity. In Example 5.15 a trace is given to demonstrate the performance of a mode switch by the protocol. A trace given in Example 5.16 shows the behaviour of the protocol during a transition to a fail-safe mode. In each case, some of the individual traces previously presented are re-used in order to eliminate the verbosity of explicitly detailing a number of intermediate steps. This also shows how these individual traces "fit together" at the protocol level.

Implicitly some properties of parallel constructs and the transition relation are employed in the following explorations. For example, imagine a number of processes are placed in parallel with each other

$$\mathbf{P}_1 \mid \mathbf{P}_2 \mid \ldots \mid \mathbf{P}_i \ldots \mid \mathbf{P}_n$$

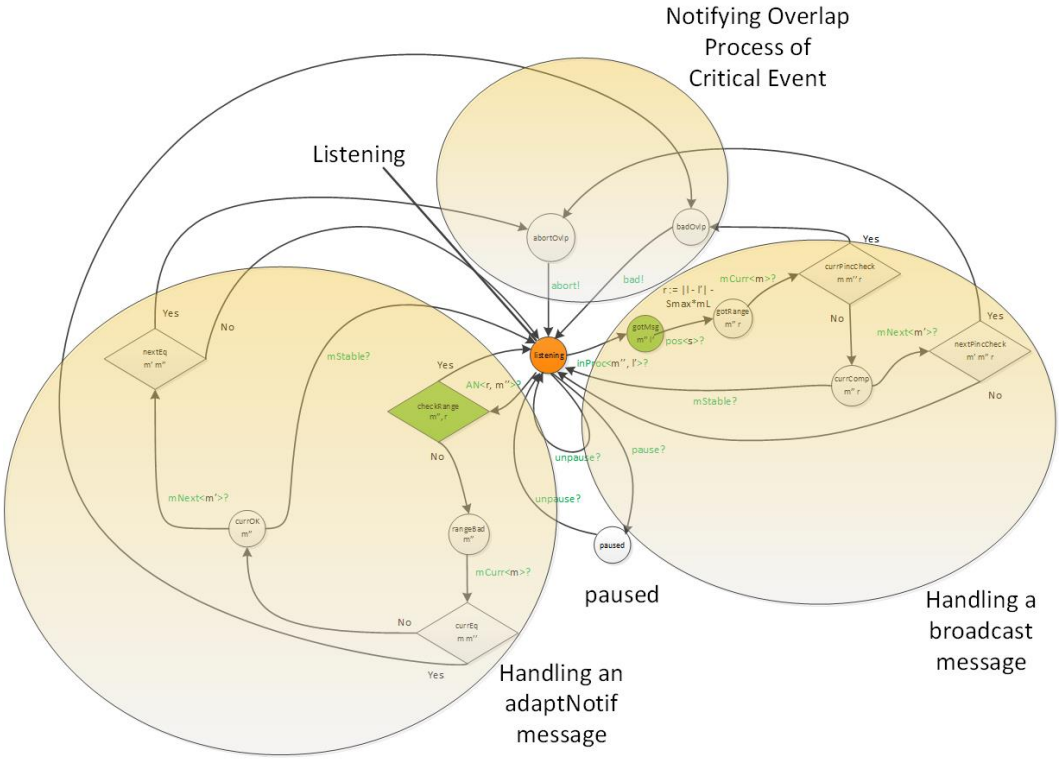


Figure 5.9: The listener process, divided into sections based on high level behaviour.

Now say one of these processes $\mathbf{P}_i$ can evolve via some discrete action δ to $\mathbf{P}_i'$

$$\mathbf{P}_i \xrightarrow{\delta} \mathbf{P}_i'$$

Then the entire parallel construct can also do this action with the remaining processes unchanged.

$$\mathbf{P}_1 \mid \mathbf{P}_2 \mid \ldots \mid \mathbf{P}_i \ldots \mid \mathbf{P}_n \xrightarrow{\delta} \mathbf{P}_1 \mid \mathbf{P}_2 \mid \ldots \mid \mathbf{P}_i' \ldots \mid \mathbf{P}_n$$

This is merely a consequence of the semantics of the software language and can be used to "lift" individual traces of the broadcast/overlap/listener components to the level of the overall protocol. Such lifting occurs implicitly in the following examples.

Example 5.15 (Performing a Mode Switch)**.** This trace is very similar to that of Example 5.9 in that most of the "work" involved in a mode switch is performed by the overlap process. The main difference here is the explicit consideration of the other processes and the synchronisations that occur along the way. The trace begins with the protocol process.

$$\textbf{sleeping} \mid \textbf{dormant} \mid \textbf{listening}$$

Now, after a number of steps given in Example 5.9, the overlap process can reach a state that is ready to output on the channel *cPause*.

$$\begin{aligned} &cAbort?.\textbf{ovAbort} + cBad?.\textbf{tfsStart} + \varepsilon(6).chanPos\langle l\rangle?.\textbf{ovReady}(m', 3, l) \\ &\quad + \boxed{\varepsilon(0).cPause!.\textbf{switchBc}(m')} \xrightarrow{cPause!} \textbf{switchBc}(m') \end{aligned}$$

Now, the listener process is capable of inputting on that same channel.

$$\textbf{listening} \xrightarrow{cPause?} \textbf{paused}$$

Thus the entire protocol process is capable of a synchronisation from overlap to listener.

$$\begin{aligned}&\textbf{sleeping} \mid \mathit{cAbort}?.\textbf{ovAbort} + \mathit{cBad}?.\textbf{tfsStart} + \varepsilon(6).\mathit{chanPos}\langle l\rangle?.\textbf{ovReady}(m',3,l)\\ &\quad + \varepsilon(0).\mathit{cPause}!.\textbf{switchBc}(m') \mid \textbf{listening} \xrightarrow{\tau} \textbf{sleeping} \mid \textbf{switchBc}(m') \mid \textbf{paused}\end{aligned}$$

From here the overlap process is capable of the transition

$$\textbf{switchBc}(m') \xrightarrow{\mathit{cWake}\langle m'\rangle!} \textbf{switchCurr}$$

Now the sleeping process is capable of being woken up by an input on the same channel

$$\textbf{sleeping} \xrightarrow{\mathit{cWake}\langle m'\rangle?} \textbf{bcWait}(m',0)$$

and so the entire process is again capable of a synchronisation, this time from overlap to broadcast.

$$\textbf{sleeping} \mid \textbf{switchBc}(m') \mid \textbf{paused} \xrightarrow{\tau} \textbf{bcWait}(m',0) \mid \textbf{switchCurr} \mid \textbf{paused}$$

From this point the overlap component is capable of reaching the state **switchListen**via an output on the channel *chanCurr*. This transition does not contribute to a synchronisation among software components but rather from the software component to the mode state. Notice that in this case the overlap transition is "lifted" to the level of the overall protocol process.

$$\textbf{bcWait}(m',0) \mid \textbf{switchCurr} \mid \textbf{paused} \xrightarrow{\mathit{chanCurr}!} \textbf{bcWait}(m',0) \mid \textbf{switchListen} \mid \textbf{paused}$$

Finally, the listener component is unpaused and the mode switch is complete. The unpausing of the listener involves a synchronisation from the overlap component to the listener component.

$$\textbf{bcWait}(m',0) \mid \textbf{switchListen} \mid \textbf{paused} \xrightarrow{\tau} \textbf{bcWait}(m',0) \mid \textbf{dormant} \mid \textbf{listening}$$

From here the broadcast process can begin periodic message broadcasting.

▲

Example 5.16 (Transition to Fail-Safe Mode)**.** In this example, a "bad" event is shown to occur, after which a reaction to a fail-safe mode is initiated and completed. The "bad" event corresponds to either a reaction to a notification message as per Example 5.13 or a reaction to an incoming message from another entity as per Example 5.14. It does not really matter which one is chosen for this example, so let us say the former occurs. Instead of starting with the protocol process this time, let's assume that a mode switch has occurred as per Example 5.15 and that the broadcast process has been broadcasting for some time, and is in the state **bcWait** with mode parameter m and time parameter t. Hence the process we start with is

$$\textbf{bcWait}(m,t) \mid \textbf{dormant} \mid \textbf{listening}$$

In a number of steps detailed in Example 5.13, the listener component will eventually reach the state **currEq**.

$$\textbf{bcWait}(m,t) \mid \textbf{dormant} \mid \textbf{listening}\ldots \xrightarrow{\mathit{chanCurr}\langle m\rangle?} \textbf{bcWait}(m,t) \mid \textbf{dormant} \mid \textbf{currEq}(m,m'')$$

Now we are at a point where the listener is ready to output on the channel *cBad* and transition back to the state **listening**. From Example 5.12 we can see that the overlap component is ready to input on that channel, transitioning to the state **tfsStart**. Hence the entire process can synchronise, performing a τ action.

$$\textbf{bcWait}(m,t) \mid \textbf{dormant} \mid \textbf{currEq}(m,m'') \xrightarrow{\tau} \textbf{bcWait}(m,t) \mid \textbf{tfsStart} \mid \textbf{listening}$$

Eventually, as per the steps outlined in Example 5.12, the overlap component can reach the state **tfsBc**, updating the mode state with the new mode in the meantime. From here, it is ready to notify the broadcast process of the fail-safe transition on the channel $cTrans$

$$\textbf{tfsBc} \xrightarrow{cTrans!} \textbf{tfsListen}$$

In turn, the broadcast process can input on the channel $cTrans$ and retreat to the sleeping state

$$\textbf{bcWait}(m,t) \xrightarrow{cTrans?} \textbf{sleeping}$$

Combining these complementary transitions in the overall process, we get

$$\textbf{bcWait}(m,t) \mid \textbf{tfsBc} \mid \textbf{listening} \xrightarrow{\tau} \textbf{sleeping} \mid \textbf{tfsListen} \mid \textbf{listening}$$

Now, one more synchronisation is needed, from the overlap component to the listening component on the channel $cUnpause$. This ensures that the listening process is not in the midst of a reaction to a message or a coverage notification. The self loop on the channel $cUnpause$ in the listening state and the transition from the process **tfsListen** in the overlap component allows for this synchronisation

$$\textbf{sleeping} \mid \textbf{tfsListen} \mid \textbf{listening} \xrightarrow{\tau} \textbf{sleeping} \mid \textbf{dormant} \mid \textbf{listening}$$

Now the transition to a fail-safe mode is complete, with the mode state updated and the broadcast component notified and sent to a sleeping state. ▲

5.4 Summary of Protocol

In summary, the protocol process is a term in the software language designed to control entities and guarantee their safe behaviour. The process can be divided into three sub-components: a broadcast component, an overlap component and a listener component. The three components are combined together in parallel i.e. they all operate concurrently. They interact with each other and with the other entity components via message passing, and incorporate real time into their design also. The following briefly re-visits the function of each of the protocol components and Remark 5.17 contains a short discussion on the wider context within which the protocol is housed.

The purpose of the broadcast process is to periodically notify its environment of the current mode of operation. This is known in the literature as beaconing. Any entities within coverage of an entity that is beaconing will receive its messages and can then adapt their behaviours accordingly. The periodicity is achieved via the state **bcWait** which has a timer that gets reset every time a message is sent and is then left to time back out to 0 before the next message can be sent.

The next process is called the overlap process. The primary purpose of this process is to enable mode switches. In a similar manner to the broadcast process, this process beacons with an upcoming mode as the entity prepares to switch to this mode; this occurs concurrently with the beaconing of the broadcast process. After a sufficient time, the mode switch is finalised by this process through a number of communications with various other components e.g. the mode state. On top of this, the overlap component can react to signals sent by the listener component to abort the current change of mode or to effect a switch to a fail-safe mode.

The listener process responds to two types of event: messages from other entities and coverage notification messages. In both cases, analysis is done and a decision is made to either signal an abort, signal a fail-safe transition or simply do nothing. If sent, the signals are received by the overlap process, which reacts accordingly.

The traces presented in this section explore these three components in terms of their various behaviours. This is useful for strengthening intuitions relating to the purpose of each component. However, traces alone

are not enough to guarantee safety. For this, a mathematical proof is needed. Now that the protocol has been covered, we are ready for such a proof. This is done in Chapter 6.

Remark 5.17 (Wider Context for Protocol)**.** Process traces are implicitly assumed to exist within the wider context of the other protocol processes and the components of the containing entity. Explicit inclusion of these other components in all the traces would amount to an excess of detail; hence they are omitted. However, it is useful to be aware of these external components. Certainly in the overall proof of safety they cannot be ignored. To be more specific, aside from the protocol components, there is a mode state and an interface, as well as a position vector in any given entity. For a schematic diagram of an entity showing these components and how they are interconnected, see Figure 4.1. ▲

Chapter 6

Safety Proof

This chapter presents a proof of safety of the Comhordú protocol. The presentation given here is what might be called "semi-formal". The aim of such a presentation is to give an intuitive explanation of how the protocol achieves safe behaviour and to consolidate that intuition with something a little more structured and formal. Strictly speaking though, this is not a formal proof. It is rather a more human readable description of the development in Coq, with low levels of detail omitted. At the time of writing, the Coq model of Comhordú can be found online at: https://www.scss.tcd.ie/ bhandalc/CoqDoc/toc.html. Hence, the primary goal here is understanding as opposed to rigour. The chapter is broken down as follows: Section 6.1 gives a brief explanation of how safety emerges from the protocol; Section 6.2 discusses and presents a number of auxiliary definitions that are introduced in order to support the proof; Section 6.3 presents a selection of the theorems comprising the proof of safety.

6.1 Safety at a Glance

Roughly speaking, system safety in a Comhordú model means that there are no pairwise incompatibilities between entities. That is, in any state of the system, there must not exist any pair of entities which are "too close together" relative to their respective modes. The question briefly addressed here is: how does Comhordú guarantee safety? In other words, if a system were to be arranged such that all its constituent entities were operating according to Comhordú, how does adherence to the protocol ensure that there is no reachable state of the system in which there is an incompatibility?

To answer this question, let's first briefly review how the protocol works. For a more detailed explanation of this, please refer to Chapter 5. Recall that the protocol is a parallel composition of three components: the broadcast component, the overlap component and the listener component. The broadcast component is quite simple: it periodically transmits messages containing the current mode to the environment. This is known in the distributed systems literature as beaconing.

The overlap component is more complex. It also beacons to its surrounding entities, but only when a mode switch is taking place. The mode beaconed is the mode into which the entity hopes to transition, a.k.a. the next mode. Along with this, the overlap component receives messages from the listener component which may cause it to abort a mode transition or, if necessary, to initiate a transition to a fail-safe mode. Finally, the listener component listens to the environment for messages from other entities and also for notifications of coverage for messages that were sent previously by this entity. The listener then analyses these messages/notifications and, if the messages are incompatible or the coverages insufficient, signals this to the overlap component which reacts accordingly.

Now we are in a position to discuss system safety. Roughly speaking, safety emerges as follows. Whenever an entity wants to transition to a mode, it must first broadcast that mode for a set period of time to the surrounding environment. Enough time is left for entities in the vicinity of the broadcasts to react to the messages by

adapting their behaviour accordingly. Assuming that all possibly incompatible entities react by transitioning to fail-safe modes, by the time this time period is over, the area surrounding the broadcasting entity is safe relative to the mode in question. The entity is then ready to transition to this new mode, and does so. Safety is then maintained while in that mode via beaconing i.e. by repeatedly broadcasting that mode to a sufficient neighbourhood. When a mode switch to another mode is desired, the whole process starts again, and for a period the entity is concurrently broadcasting both its current and next modes until the next mode is secure and the mode switch takes place. Now, since we assumed an arbitrary entity, and have roughly demonstrated that it is never the cause of an incompatibility, then we can generalise and conclude that all pairs of entities are compatible.

Two events beyond the control of the entity may occur, forcing it to adapt its behaviour. The first is a degradation in coverage. What this means is that the entity's messages do not reach a sufficiently large surrounding area. If this happens to messages of the current mode, the entity must begin a transition to a fail-safe mode immediately, since it cannot assume that other entities in the "danger zone" have received its messages. If the degradation is for next-mode messages, then the entity simply aborts its current attempt at a mode transition for now. The second external event that may affect an entity is the receipt of a message from another entity. The contents of such a message will include the position and mode of the sending entity. The receiving entity must decide then, based on this information, if the position and mode received are possibly incompatible with its own position and mode, correcting for possible movement since the message was sent. If an incompatibility occurs, then behaviour is adapted much as in the case of the coverage degradation event: a transition to a fail-safe mode follows an incompatibility with the current mode, while a cancellation of the pending mode transition follows an incompatibility with the next mode.

The actual proof of safety differs somewhat from the high level description given here. This is because the notion of a zone of entities surrounding a particular entity is cumbersome to formalise. Instead, the proof is approached with the strategy of "reductio ad absurdum". For this, first it is assumed that there is a pair of incompatible entities. Then a contradiction is derived, involving the fact that one of the entities should have reacted to a message from the other by now. Hence it is possible to reject the possibility of such an incompatibility. The remainder of this chapter is geared towards this proof by contradiction.

6.2 Auxiliary Definitions

Before moving on to the proof of safety, a number of definitions need to be established. These are referred to as "auxiliary definitions" in contrast to the standard definitions comprising the syntax and semantics of the underlying languages and the protocol specification. What sets these auxiliary definitions apart from the standard ones is that their sole purpose is to provide a framework in which a proof of safety may be formulated: they are not essential ingredients of the protocol itself. They have arisen from natural language descriptions of the properties of the protocol with respect to its adherence to safety. They can be considered in some way to be a formalisation of many of the underlying intuitions behind the proof of correctness.

The majority of these definitions take on the form of what we will refer to here as "history relations". Given a reachable state (see Definition 6.2) in the LTS of the system, we know from the definition of reachability that there is some finite trace starting at an initial state and ending at this one. It is then possible to think of properties that have held across this trace. For example, it might be for that the final ten states of the trace, including this one, the entity at index 12 was in mode m. It happens that statements of a similar nature to this one form much of the reasoning that comprises the safety proof. The question then is, how do we formalise such statements?

An initial attempt at a formalisation of these "history properties", as they were, would be to define auxiliary structures called finite traces and then define a real-time logic (see Section 2.4) over these traces with operators such as until, next etc. In fact, this was the basis of an initial proof of safety that was sketched out before the current one came into being. However, the explicit use of traces and such a logic proved cumbersome, and the approach was dropped in pursuit of a more elegant idea: history relations. The basic idea behind these relations

is that they would be defined inductively and would abstract from unnecessary detail of the underlying trace. For example, rather than record the exact sequence of states leading up to this one, a relation might simply record the time since the last time the mode was switched.

However, there was one feature of these relations that was overlooked on their initial conception: the non-uniqueness of history. That is, given a certain reachable state, there is no way of determining *the* sequence of states that lead to this state, making it reachable. In fact, the semantics allow for a possibly infinite number of such histories. Equivalently, this problem amounted to the fact that there was not enough information contained within a state alone to determine a unique history for it. Immediately, it seemed like the idea of history relations would then need to be abandoned, and traces once again seemed like the best option. But then it was discovered that the relations could be "tweaked" somewhat so that they "worked". This was done by adding into the parameter list of each relation the proof of reachability of the state in question.

It seems, then, that we're back at square one: history relations were supposed to be a less cumbersome alternative to traces, but now included in every history relation's parameter list was a proof of reachability, which itself is essentially a finite trace, starting at an initial state. However, in hindsight, the verbosity of the trace/real-time logic method arose due to an explicit infinite characterisation of traces, coupled with a complex real-time logic including future operators. In contrast, the approach using the history relations relies on implicit finite traces and there is no mention of future states. Also, the inductive characterisation of these relations makes them amenable to proofs by induction, whereas the explicit, infinite trace approach was based on co-induction, making them more difficult to manipulate. Thus, the history relations approach still proves to be the better alternative, and in fact the inclusion of the reachability proofs hardly adds anything to their complexity.

Now, before moving on to the auxiliary definitions themselves, let us give a brief explanation of the structure of these definitions. Most will consist of some prose explanation of the defined object followed by a number of inductive rules. The rules will be of the form $\frac{A \qquad B}{C}$. Each such rule constitutes an inductive constructor of the corresponding relation; all terms of the relation are then built from a finite application of these constructors alone. The meaning of such a rule is that whenever terms of types A and B can be constructed, then a term of type C can be constructed. Equivalently, to shy away from the underlying constructivism, such a rule can be interpreted logically as saying that whenever A is true and B is true, then so is C. A and B may be referred to as premises or hypotheses. Alternatively, they can be seen as argument types of a constructor function. In Coq, by the Curry-Howard isomorphism, these are two sides of the same coin. This schema can easily be extended to include an arbitrary number of premises. Now, let us present these definitions.

6.2.1 Some Basic Definitions

Before the history relations are introduced, a few basic notions are defined here.

Definition 6.1 (Initiality)**.** A network is said to be initial if every constituent entity is in a fail-safe mode and is also "running" the protocol process as its software component. ■

Definition 6.2 (Reachability)**.** Reachability is the reflexive transitive closure of the delay and discrete transition relations upon the set of initial states. That is, an initial state is reachable and also any state that can be reached via a discrete or timed transition from a reachable state is also reachable. Proofs of reachability then correspond to finite traces, beginning with an initial state and finishing with the state that is deemed reachable.

In the following, the functions *reachNetDel* and *reachNetDisc* construct a proof of reachability for some state given that a "previous" state was reachable and given the transition between the states. For example, $reachNetDisc(\mathbf{N}, \mathbf{N}', \delta, p, w)$ constructs a proof of reachability for $\mathbf{N}'$ given the proof p of reachability for the network $\mathbf{N}$ and the transition between the networks w; the variable δ is just the label on the action linking the states. Nested applications of the constructor can be unwound, revealing the essential trace structure of these proofs. ■

Definition 6.3 (Buffering Predicates)**.** A number of similar predicates are defined to capture the buffering of messages by the interface components of entities. Theoretically speaking, this proof development could proceed without such predicates. However, they help reduce verbosity. The names of these predicates are *incoming*, *incomingNotif* and *outgoing*. Each predicate records the fact that a certain message has been in the input/notification/output list, respectively, of the interface component for some entity in a network. Except for *incoming* the timestamp of this message is also recorded by the relation.

Below are the rules for constructing these relations. Since they are all similar, the only one discussed here is *outgoing*. The first premise asserts the existence of an entity at some position in the network. The entity in question is deconstructed into its various components: software process, position, interface and mode state. The interface is in turn deconstructed into a triple of lists. The second premise then simply says that an element $\langle \vec{v} \rangle_t$ is in the output list. From these premises, *outgoing* is drawn as the conclusion. This relation acts as a wrapper, implicit in which is the existence of the entity and the element in the output list of its interface. Such existential notions would be cumbersome to work with, hence the need for such a wrapper. The relations *incoming* and *incomingNotif* are of an analogous nature.

$$\frac{\lhd \mathbf{P}, l, (\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}), \mathbf{K} \rhd @i. : \mathbf{N} \qquad \langle \vec{v} \rangle_0 : \mathbf{L_i}}{incoming(\vec{v}, i, \mathbf{N})}$$

$$\frac{\lhd \mathbf{P}, l, (\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}), \mathbf{K} \rhd @i. : \mathbf{N} \qquad \langle \vec{v} \rangle_t : \mathbf{L_o}}{outgoing(\vec{v}, t, i, \mathbf{N})}$$

$$\frac{\lhd \mathbf{P}, l, (\mathbf{L_i}, \mathbf{L_o}, \mathbf{L_n}), \mathbf{K} \rhd @i. : \mathbf{N} \qquad \langle \vec{v} \rangle_t : \mathbf{L_n}}{incomingNotif(\vec{v}, t, i, \mathbf{N})}$$

■

Definition 6.4 (Path Predicates)**.** Here we define a number of predicates that capture various behaviours of the listener component of the protocol. Recall from Chapter 5 that the listener component processes both incoming notification messages and incoming messages from other entities. Now, such messages may at times be ignored i.e. a notification of a sufficiently covered message or an incoming beacon from an entity that is sufficiently far away in order to be compatible. Otherwise, the receiving entity reacts to the message in some way. There are two such reactions: transitioning to a fail-safe mode or aborting the current mode transition. The former occurs when an incompatibility or insufficiency occurs in relation to the current mode; the latter is similar but relates to the next mode. Now, in order to process these reactions, the listener component must traverse some path of states, before finally notifying the overlap component of the action that is to be taken. Since there are two reactions, and two types of message, there are four such paths. *msgBadPath* and *msgAbortPath* are predicates asserting that an entity is reacting to an incompatibility with the current or next mode respectively. *notifBadPath* and *notifAbortPath* assert that the entity is reacting to a coverage degradation notification, again for the current or next mode respectively.

For the full definitions of each of these predicates, please refer to the "EntAuxDefs.v" file of the Coq model. Here, in order to give a flavour of what these predicates are like, we will just define one of these predicates, namely *msgBadPath*. The first rule says that if the state of the listener component is *gotMsg* for some received mode and position m'' and l', and the entity is incompatible with respect to that mode, normalising for the possible variance in the received position, then *msgBadPath* holds. The next rule is similar, but the state is now *gotRange* and the entity has already computed the distance r, so there is less calculation to do in the rule. The final rule includes the state *badOvlp* in the predicate: this state is waiting to send a "bad" message to the overlap component. The other predicates are similar, except dealing with either the signalling of an abort, or the processing of notification messages, or both. Notice that these predicates capture a composite property of both the mode state and the software process, and so cannot be defined any lower than the entity level.

$$\frac{gotMsgState(m'', l', \mathbf{P}) \qquad currMode(\mathbf{K}) = m \qquad pincv(m, m'', (|l - l'| - s_{max} * msgLatency))}{msgBadPath(\lhd \mathbf{P}, l, \mathbf{I}, \mathbf{K} \rhd,)}$$

$$\frac{gotRangeState(m'',r,\mathbf{P}) \qquad currMode(\mathbf{K}) = m \qquad pincv(m,m'',r)}{msgBadPath(\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd,)}$$

$$\frac{badOvlp(\mathbf{P})}{msgBadPath(\lhd\mathbf{P},l,\mathbf{I},\mathbf{K}\rhd,)}$$

■

Definition 6.5 (State Predicates)**.** It is often necessary to look within an entity in order to determine the state of its software component. For this reason, a number of state predicates are defined. The entities of interest for which these predicates are defined are reachable, containing processes that are triples of broadcast, overlap and listener components. As such, the state predicates implicitly assert that the process in question is such a triple. They then drill into this process triple and mention properties about one or more of the sub-components. They can be classified under two broad categories: simple and compound.

Simple state predicates roughly correspond to the protocol definitions. Each predicate asserts that the software component is in the "shape" of a particular protocol process. For example, the predicates *switchListen* and *switchCurr* refer to the middle component of a software term, namely the overlap component. Please refer to Chapter 5 for more on these definitions.

Compound state predicates build on the simple state predicates, thereby achieving characterisations of more complex or more abstract properties. For example, the predicate *nextSince* captures the fact that a process is in one of a number of states.

■

Definition 6.6 ($sufficient$)**.** $sufficient(m,r)$ asserts whether the coverage r of a message containing the mode m was large enough in a certain sense. The following inequality distills this notion of "large enough" into something rigorous. It asserts that the distance r is greater than the maximum minimum distance of incompatibility for the mode in question, plus a number of "buffer" terms. This should ensure that, after a certain amount of time after the message delivery, the area surrounding the sender is in some sense "safe".

$$sufficient(m,r) \stackrel{\text{def}}{=} d_{max}(m) + 2 * s_{max} * (max(adaptNotif,trans) + period(m) + trans(m)) \leq r$$

Rather than writing $sufficient(m,r)$ each time when referring to this predicate, sometimes it will just be said that a particular message is sufficient. The parameters should, in these cases, be obvious from the context. ■

6.2.2 The History Relations

Definition 6.7 (tfs)**.** $tfs(m,t,i,\mathbf{N},p)$ says that the entity i in the network $\mathbf{N}$ has been transitioning to a fail-safe mode from the mode m for a time of t. Intuitively, this relation is used to record how long an entity has been waiting to enter a fail-safe state. The idea then is that it should be possible to prove that this waiting time is bounded from above, which allows us to conclude that all entities beginning a fail-safe transition eventually "commit" to this transition. This is an important property since the protocol relies on timely adaptations of entity behaviour to fail-safe modes if they are at risk of causing an incompatibility.

For the base case of this relation, the entity must be in the state $tfsStart$, and the time parameter begins at zero. The condition is preserved by a discrete transition as long as the derivative state belongs to a certain subset of locations (state patterns), characterised by the predicate tfs. Since delay does not alter location, we do not need this stipulation in the premise of the delay case. Rather, we simply add the value of the delay to the running total of time so far.

$$\frac{p : reach(\mathbf{N}) \qquad tfsStart(i,\mathbf{N}) \qquad currModeNet(m,i,\mathbf{N})}{tfs(m,0,i,\mathbf{N},p)}$$

$$\frac{tfs(m,t,i,\mathbf{N},p) \qquad tfs(i,\mathbf{N}') \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{tfs(m,t,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{tfs(m,t,i,\mathbf{N},p) \qquad w:\mathbf{N}\xrightarrow{d}\mathbf{N}'}{tfs(m,(t+d),i,\mathbf{N}',(reachNetDel(\mathbf{N},\mathbf{N}',d,p,w)))}$$

■

Remark 6.8 (History Relations, Shared Structure)**.** Many of the history relations share a similar structure. There are usually three constructors: a base case constructor, a discrete inductive case and a delay inductive case. Some relations are concerned with a single event that happened in the past, while others deal with a continuous property, some are a mixture of both. The base case captures either the occurrence of an event like message sending or the beginning of some property e.g. a certain state was just entered. The inductive cases then extend the relation from a past state to the present state, with the delay constructor updating the time variable appropriately. Taking the tfs relation just presented as an example, it deals with the continuous property of having been in one of the tfs states for the past t units of time. The base case covers the first instance in which this property is true: entering the $tfsStart$. The inductive constructors then allow the relation to be extended given that the derivative state belongs to a select group of states, characterised by tfs. ▲

Definition 6.9 (*pendingNet*)**.** $pendingNet(i,\mathbf{N},p)$ is a precursor to the *nextSince* relation of **Definition 6.10**. It is needed in order to "normalise" the relationship between the time parameter of the waiting state and that of *nextSince*, for initially there is a corner case in which there is an extra $period(m)$ added to the waiting time parameter. *pendingNet* only lasts for two states, until this corner case has been overcome, and then it hands "control" over to *nextSince*. Note that there are no delay constructors. This is because it should be provable that a state cannot delay when it is *pendingNet*.

$$\frac{p:reach(\mathbf{N}) \qquad init(m,i,\mathbf{N}) \qquad ovWait(m,t.x,y,i,\mathbf{N}') \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{pendingNet(i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{pendingNet(i,\mathbf{N},p) \qquad ovWait(m,t.x,y,i,\mathbf{N}) \qquad ovWait(m,t.x,y,i,\mathbf{N}') \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{pendingNet(i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{pendingNet(i,\mathbf{N},p) \qquad ovWait(m,t.x,y,i,\mathbf{N}) \qquad ovReady(m,t,l,i,\mathbf{N}') \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{pendingNet(i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{pendingNet(i,\mathbf{N},p) \qquad ovReady(m,t,l,i,\mathbf{N}) \qquad ovReady(m,t,l,i,\mathbf{N}') \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{pendingNet(i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

■

Definition 6.10 (*nextSince*)**.** $nextSince(m,t,i,\mathbf{N},p)$ roughly says that entity i in the reachable network $\mathbf{N}$ has initiated a mode transition to m and has been broadcasting mode m messages since t time units ago. It also implicitly records that since initiating this transition, the entity has not changed its mode state. The purpose of this relation is twofold. First, it is a precursor to a similar relation for the current mode, *currSince*. Second, it allows certain properties to be expressed and proved e.g. whenever it holds, we expect that there has been periodic messages sent since the relation began to hold.

The base case says that if between states an entity changes from *ovReady* to *ovWait*, with parameter m, then it must have just begun the mode transition to m, and so the relation holds with time parameter 0.

The discrete inductive case states that when an entity has been *nextSince m* in a certain network, and in a derivative network, it is still in a *nextSince*, then it is still *nextSince* in the derivative network. The time parameter has not changed because the action was discrete.

The timed case is similar except time is updated by adding the delay to it. Also, there is no need for the assertion of *nextSince* for the derivative state- a general result should be that states are preserved by the delay relation. Notice that reachability is implicitly carried forward from *pendingNet*, and so does not need explicit

mention in these rules. Notice also in the base case that t' is free with respect to t, allowing for the time to be updated.

$$\frac{pendingNet(i,\mathbf{N},p) \qquad ovReady(m,t,l,i,\mathbf{N}) \qquad ovWait(m,t.x,y,i,\mathbf{N}') \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{nextSince(m,0,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{nextSince(m,t,i,\mathbf{N},p) \qquad nextSince(i,\mathbf{N}') \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{nextSince(m,t,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{nextSince(m,t,i,\mathbf{N},p) \qquad ovWait(m,t.x,y,i,\mathbf{N}) \qquad ovReady(m,t,l,i,\mathbf{N}') \qquad w:\mathbf{N}\xrightarrow{d}\mathbf{N}}{nextSince(m,(t+d),i,\mathbf{N}',(reachNetDel(\mathbf{N},\mathbf{N}',d,p,w)))}$$ ■

Definition 6.11 (*currSince*)**.** $currSince(m,t,i,\mathbf{N},p)$ roughly says that entity i in the network $\mathbf{N}$ has been in the mode m, or transitioning to the mode m in the network $\mathbf{N}$ for the last t time units. The main purpose of this relation is to allow for the expression of theorems stating that periodic broadcast precedes any entity in a non-fail-safe mode. This should become apparent when the results are discussed later on. Note, sometimes the last three arguments of this relation are omitted when they are obvious from the context.

In the base case of this relation, an entity in a network is deemed to be $currSince(m,t)$ if a predecessor network had that same entity as $nextSince(m,t)$, and in the process of the transition the entity has transitioned from the state *switchCurr* to *switchListen*. It can be shown that this transition implies that the mode m must be sent to the mode state, causing a mode switch. Notice that the time parameter t is passed along from one relation to the next in the fashion of a bat from one member of a relay team to the next. This is essential in proving properties of *currSince* that carry over from *nextSince*. The inductive cases are simple, and follow the standard pattern shared by much of the history relations. It may be worth noting that there is no need to explicitly enforce reachability here because it will follow from *nextSince*.

$$\frac{nextSince(m,t,i,\mathbf{N},p) \qquad switchCurr(i,\mathbf{N}) \qquad switchListen(i,\mathbf{N}') \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{currSince(m,t,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w))))}$$

$$\frac{currSince(m,t,i,\mathbf{N},p) \qquad currModeNet(m,i,\mathbf{N})' \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{currSince(m,t,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{currSince(m,t,i,\mathbf{N},p) \qquad w:\mathbf{N}\xrightarrow{d}\mathbf{N}}{currSince(m,(t+d),i,\mathbf{N}',(reachNetDel(\mathbf{N},\mathbf{N}',d,p,w)))}$$ ■

Definition 6.12 (*sent*)**.** $sent(\vec{v},t,i,\mathbf{N},p)$ is the relation capturing the sending of a message from the software component to the interface component. The base case is the only case of any interest here. It says that $\vec{v}$ was sent in this instant (0 time units ago) by entity i when the entity at i can synchronise across the channel *chanOutP* from its software component to its interface component. One thing to note is that sent, like many of these history relations, is preserved by both transitions, as would be expected. That is, if a message has been sent at some point, then it remains sent at a later point. Also, at this later point, the time parameter will not have decreased- though it may have increased. In the following, the expression $\mathbf{E}@i.:\mathbf{N}$ means that the entity $\mathbf{E}$ is located at index i in the network $\mathbf{N}$: recall that a network is modelled as a list. Also, due to spatial limitations, there is an implicit assumption of the following equalities in the first rule:

$$\mathbf{E} \stackrel{\text{def}}{=} \lhd \mathbf{Q},l,\mathbf{I},\mathbf{K} \rhd \qquad\qquad \mathbf{E}' \stackrel{\text{def}}{=} \lhd \mathbf{Q}',l',\mathbf{I}',\mathbf{K}' \rhd$$

$$\frac{p:reach(\mathbf{N}) \qquad \mathbf{E}@i.:\mathbf{N} \qquad \mathbf{E}'@i.:\mathbf{N}' \qquad \mathbf{Q}\xrightarrow{chanOutP\langle\vec{v}\rangle!}\mathbf{Q}' \qquad \mathbf{I}\xrightarrow{chanOutP\langle\vec{v}\rangle?}\mathbf{I}' \qquad w:\mathbf{N}\xrightarrow{\tau}\mathbf{N}'}{sent(\vec{v},0,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\tau,p,w)))}$$

$$\frac{sent(\vec{v},t,i,\mathbf{N},p) \qquad w:\mathbf{N}\xrightarrow{\delta}\mathbf{N}'}{sent(\vec{v},t,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{sent(\vec{v},t,i,\mathbf{N},p) \qquad w : \mathbf{N} \xrightarrow{d} \mathbf{N}}{sent(\vec{v},(t+d),i,\mathbf{N}',(reachNetDel(\mathbf{N},\mathbf{N}',d,p,w)))}$$ ■

Definition 6.13 (*delivered*)**.** $delivered(\vec{v},r,t,i,\mathbf{N},p)$ says that the message $\vec{v}$ was delivered to the radius r t time units ago by the entity i in the network $\mathbf{N}$. For any entity existing in a reachable state, this relation "records" all the messages that have been delivered by that entity since the initial state.

$$\frac{p : reach(\mathbf{N}) \qquad w : \mathbf{N} \xrightarrow{\langle\vec{v}\rangle^{l}_{r}!_{i}} \mathbf{N}'}{delivered(\vec{v},r,0,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\langle\vec{v}\rangle^{l}_{r}!_{i},p,w)))}$$

$$\frac{delivered(\vec{v},r,t,i,\mathbf{N},p) \qquad w : \mathbf{N} \xrightarrow{\delta} \mathbf{N}'}{delivered(\vec{v},r,t,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{delivered(\vec{v},r,t,i,\mathbf{N},p) \qquad w : \mathbf{N} \xrightarrow{d} \mathbf{N}}{delivered(\vec{v},r,(t+d),i,\mathbf{N}',(reachNetDel(\mathbf{N},\mathbf{N}',d,p,w)))}$$ ■

Definition 6.14 (*received*)**.** $received(\vec{v},t,i,\mathbf{N},p)$ says that the message $\vec{v}$ was received t time units ago by the entity i in the network $\mathbf{N}$.

$$\frac{p : reach(\mathbf{N}) \qquad w : \mathbf{N} \xrightarrow{\delta} \mathbf{N}' \qquad \mathbf{E}@i. : \mathbf{N} \qquad \mathbf{E}'@i. : \mathbf{N}' \qquad \mathbf{E} \xrightarrow{\langle\vec{v},l,r\rangle?} \mathbf{E}'}{received(\vec{v},0,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{received(\vec{v},t,i,\mathbf{N},p) \qquad w : \mathbf{N} \xrightarrow{\delta} \mathbf{N}'}{received(\vec{v},t,i,\mathbf{N}',(reachNetDisc(\mathbf{N},\mathbf{N}',\delta,p,w)))}$$

$$\frac{received(\vec{v},t,i,\mathbf{N},p) \qquad w : \mathbf{N} \xrightarrow{d} \mathbf{N}}{received(\vec{v},(t+d),i,\mathbf{N}',(reachNetDel(\mathbf{N},\mathbf{N}',d,p,w)))}$$ ■

6.3 Results

In this section the main results comprising the safety proof are presented. These are select theorems and lemmas deemed to be of significant interest or importance such that they are included here. This is by no means an exhaustive presentation of the entire collection of all the results from Coq. Rather, it can be thought of as the "highlights" from that development. Each result presented, and indeed many results that were omitted, are interlinked in a structure culminating at the top level in the proof of safety. Such a structure is often referred to as a "proof tree". A more correct term would be "proof graph" since it is possible for one result at a low level to contribute directly to many results at higher levels; however "tree" is a useful way of thinking about the graph because it implies acyclicity, an essential property of any proof.

In any case, there are two opposing ways in which such a graph can be expounded. On the one hand, one could explain the most important result first and then work down towards lower level results. Alternatively, the lowest level results could be put forth to begin with, eventually building up to the top level result in the end. Both have their merits. The "top down" approach is goal driven, allowing a reader to understand the theorem in as much or as little depth as is needed. However, often while a logical understanding can be gained from such an approach, i.e. "A is true because B, C and D are true", a deeper intuitive understanding is often lacking because this is often hidden in the machinery at the lower levels. On the other hand, starting from a low level, a reader has a full understanding of each theorem as it is encountered. However, there is a lot of trawling through results required before the main results are reached. Of course, a "middle out" approach is also possible, starting with some intermediate level results and working either up or down from there. But unless there is good reason for doing so, this seems like an arbitrary choice.

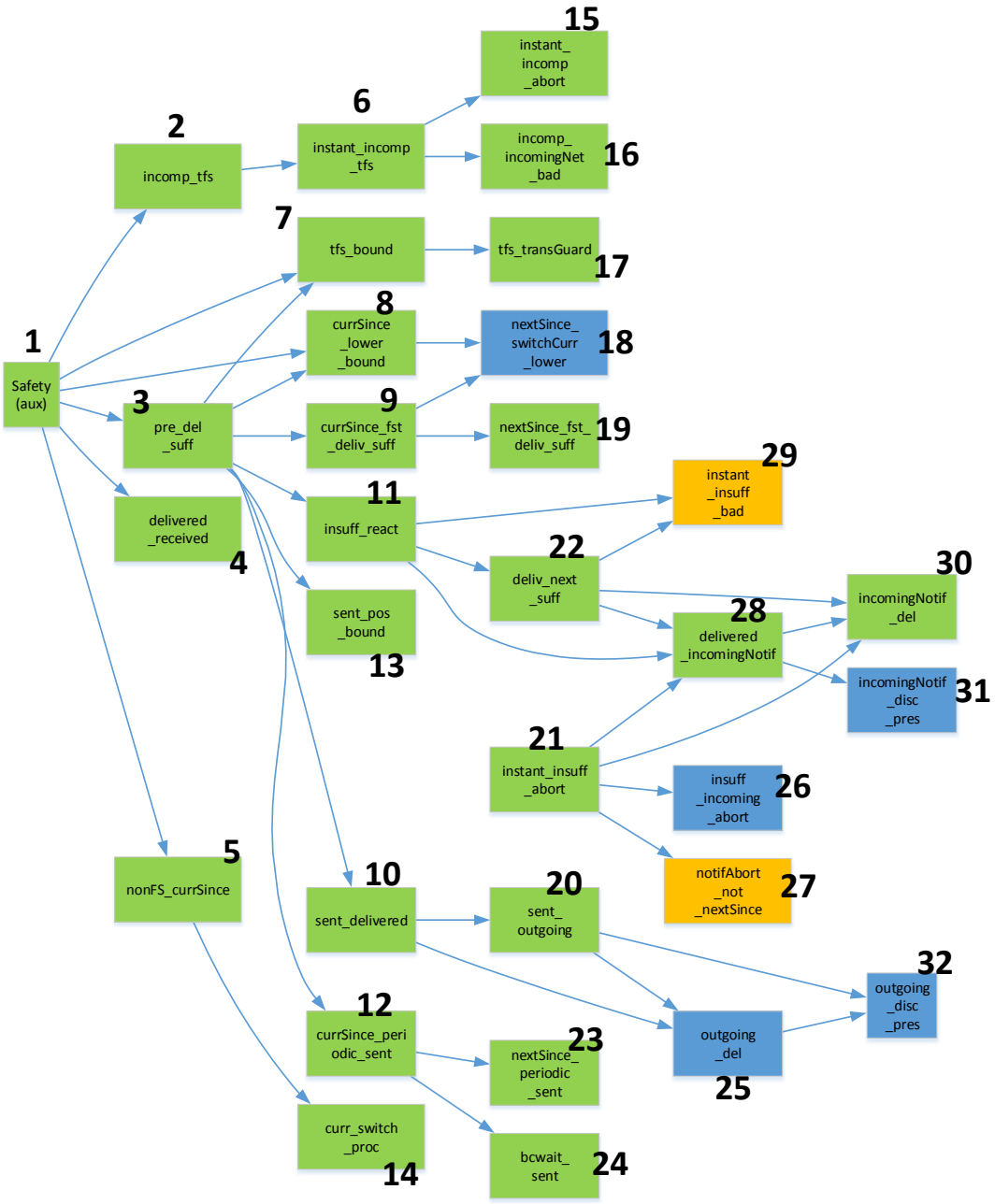


Figure 6.1: The proof map. An arrow from box A to box B indicates a dependency of A on B i.e. the result B is used in the proof of A. The most significant results are further to the left while the lower level results are to the right.

In this section, the top down approach will be taken starting with the proof of safety in terms of its immediate "children" in the graph, and then working down from there in a breadth-first manner. This does not necessarily mean that top down is the "correct" order in which to work through the material. The reader is encouraged to follow whichever path they feel will give them the best intuitive understanding of the proof, skipping over whatever seems obvious or irrelevant, and jumping to results that might seem interesting: sometimes when reading a proof it is desirable to plunge down a few levels in depth-first fashion until there is an "aha" moment. To aid with all of this, then, a diagram illustrating the proof graph is first given in Section 6.3, and the theorems follow this. The diagram is referred to here as a "proof map": a means of navigating the collection of results comprising the overall proof which would otherwise be unwieldy. Note that the actual map is laid out from

left to right rather than from top to bottom: this was simply the best possible fit. Each result in the map is represented as a box containing the name of the theorem; beside the box is the relative[1] number of the theorem as it appears in this chapter.

Before moving on to the actual theorems, let us first briefly explain how they are presented in the following. Each theorem consists of a prose statement of what the theorem says accompanied by some remarks as to why the theorem is true and/or why it is important in the big picture.

Some theorems deemed interesting or important are followed by proofs. Each proof is a logical argument explained through prose. It will usually follow the structure of the Coq proof, which in most cases will exist, but will be condensed for human readability. Some proofs will not have been done in Coq and will be marked as such by the symbol † in their statement. In these cases, the proof outlined is a sketch or a strategy based on the intuitions as to why the theorem is true.

The level to which a proof is condensed depends on a number of factors: how interesting it is, how obvious it is, whether there exists another analogous proof. The idea is to minimise as much as possible the verbosity of the proofs, allowing only the main intuitions to shine through. In many instances, though, it is not obvious to what extent a proof should be expounded, and it may well be that the level of detail for some proofs may seem insufficient. In such a case, the interested reader is referred to the full proof in Coq, which is documented with comments. On the other hand, the proofs given here might well seem superfluous, in which case the reader is urged to skim through them, or even skip over them entirely. In particular, the induction on history relations is a commonly recurring theme and may get a bit repetitive. It is now time to present the main result, Theorem 6.1.

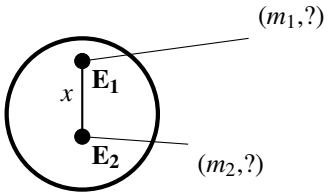
Theorem 6.1 (Safety)**.** *This is the main result stating the correctness of the protocol. It says that if a network is reachable, then it is safe. To recap, what this means is that, given a state in which all the entities are fail-safe and running the protocol process as their software component, if this state is allowed to evolve via any finite number of delays and discrete actions as per the transition semantics, then the resulting state is safe. Safety itself is defined elsewhere, but basically means that all pairs of entities in the network are separated by a sufficient distance to ensure that they are not incompatible according to their respective modes.*

Proof of Safety. Proceed by reductio ad absurdum. That is, assume the existence of a single pair of entities violating safety and deduce a contradiction. So the assumption is that there exists an initial reachable network in which there is some pair of entities that are separated by a distance less than the minimum distance of compatibility for their respective modes. Let's call the entities $\mathbf{E_1}$ and $\mathbf{E_2}$ and modes m_1 and m_2 respectively. Let x be the distance between the two entities. Then the assumption says $x < minDistComp(m_1, m_2)$. Let $\mathbf{E_1} = \lhd \mathbf{P}_1, l_1, \mathbf{I}_1, \mathbf{K}_1 \rhd$ and $\mathbf{E_2} = \lhd \mathbf{P}_1, l_1, \mathbf{I}_1, \mathbf{K}_1 \rhd$- this is simply the decomposition of each entity into its constituent parts (zooming in so to speak).

The diagram below depicts this scenario. A large black circle represents the network, whereas the black dots within this represent entities. The line joining the dots represents the distance between them, which is labelled with x. There is a line from each entity labelling it with its mode state. The mode states, appearing as pairs of the form (m,m'), represent the current and next modes respectively. A question mark in the placeholder of one of the modes denotes that its value is irrelevant, a "wildcard" so to speak. Note that there may not be any next mode, but there is always a current mode. Included in this diagram and to to the right is a reminder of the assertion that is being made about the distance x, namely that it is less than the minimum distance of compatibility for the corresponding modes. It often appears useful to include such logical statements in diagrams as they can get lost in the text otherwise. A final note on this diagram is that the entities are arranged vertically. This is done deliberately; the horizontal dimension is reserved for time, in particular to depict the histories of certain entities, a feature not present in this diagram. These graphical conventions, among others, are used intermittently throughout the following proofs to aid with the understanding of some of the theorems.

Now, returning to the proof, it proceeds in steps as follows.

[1] For brevity, the absolute number of the theorem, including the chapter number, is not given.



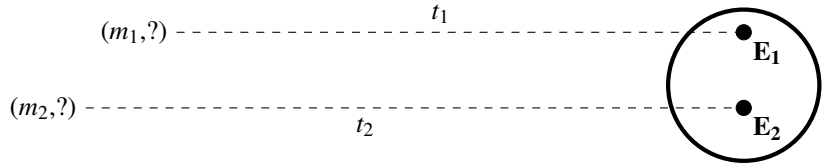
$$x < minDistComp(m_1, m_2)$$

1. Clearly both m_1 and m_2 are non-fail-safe modes because otherwise the minimum distance of compatibility would be 0, and we would have $x < 0$, which contradicts the non-negativity of distance.

2. By **Theorem 6.5**, any entity in a non-fail-safe mode can be shown to be *currSince* for some parameters. Hence both $\mathbf{E_1}$ and $\mathbf{E_2}$ have been $currSince(m_1, t_1)$ and $currSince(m_2, t_2)$ respectively.

3. Applying **Theorem 6.8** it can be shown that these values t_1 and t_2 are strictly bounded from below by $msgLatency + max(adaptNotif, trans)$.

4. Without loss of generality, we can assume that $t_1 \leq t_2$. The following picture illustrates this. A few new conventions are adopted here. The dashed lines extending from the entities from right to left represent the histories of those entities. A history is a finite sequence of states ending at the current state with all consecutive states linked by either discrete or timed transitions. It could also be referred to as a timed trace. Note that the dashed line omits the unnecessary detail of all the individual states, in tune with the abstraction from such detail that characterises most of the history relations. Labelling each history is the time corresponding to that history: this is simply the sum of all the timed transitions taken. Finally, at the end of each history are assertions of the form $(m,?)$. This signifies that the entity in question has been *currSince* in the mode m for the duration of the history. By replacing the $(m,?)$ with something else, it is possible to depict a different history relation; this is done in future proofs. Note that these notational conventions are not intended to be formal pictorial representations of the underlying maths, rather they are just an intuitive aid to the reader; hence certain "abuses of notation" such as using $(m,?)$ for both *currSince* and to represent the current mode.



5. Let us pause for a moment here to consider the direction of the remainder of the proof. The goal is to arrive at a contradiction. An overview of how this contradiction arises is as follows. The entity $\mathbf{E_1}$ is shown to have output a message at some time in the past; this is inputted by $\mathbf{E_2}$. Then it is shown that $\mathbf{E_2}$ must have reacted to that message by initiating a transition to a fail-safe mode. Since neither entity is in a fail-safe mode currently, this transition must be still taking place. Now, transitions to fail-safe mode are shown to be bounded from above by an amount *trans*, which is defined in Definition 5.3. But this bound is shown to be impossible, because it clashes with a lower bound that holds on the delivery of the message in question. Hence the contradiction arises. Let us now expand on this in more detail.

6. Here we assert the delivery of the message by $\mathbf{E_1}$. We know from **Theorem 6.3** that $currSince(m_1, t_1)$ $\mathbf{E_1}$ implies that a message of the form $\langle m_1, l' \rangle$ was delivered t time units ago to a sufficient radius (for m_1) r by $\mathbf{E_1}$. The result also gives us that t is in the interval $I_{pd}(m_1)$, $t < t_1$ and the distance between l_1 and l' is at most s_{max} *$(msgLatency + t)$. Since we have previously shown that $t_1 \leq t_2$ we can infer by transitivity that $t < t_2$.

7. We want to prove the bound $x + 2 * s_{max} * t \leq r$, showing that $\mathbf{E_2}$ was in range of $\mathbf{E_1}$ when the message delivery occurred. This is shown via some basic manipulation of inequalities. We know from the definition

of I_{pd} that: $t \leq max(adaptNotif, trans) + period(m_1) + trans(varModeIndexed1)$ so

$$2 * s_{max} * t \leq 2 * s_{max} * (max(adaptNotif, trans) + period(m_1) + trans(varModeIndexed1))$$

But we also know that $x < minDistComp(m_1, m_2)$ Adding the inequalities we get

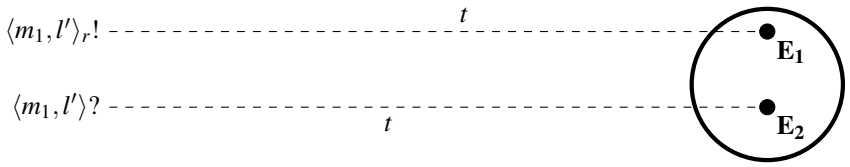
$$x + 2 * s_{max} * t < minDistComp(m_1, m_2) + 2 * s_{max} * (max(adaptNotif, trans) \quad (6.1)$$
$$+ period(m_1) + trans(varModeIndexed1)) \quad (6.2)$$

Also, we have already shown that r is sufficient for the mode m_1, which expands to:

$$d_{max}(m_1) + 2 * s_{max} * (max(adaptNotif, trans) + period(m_1) + trans(varModeIndexed1)) \leq r$$

Since $minDistComp(m_1, m_2) \leq d_{max}(m_1)$ by definition, then we can infer via transitivity that $x + 2 * s_{max} * t < r$. Which can be weakened to $x + 2 * s_{max} * t \leq r$.

8. We can now show that $\mathbf{E_2}$ received the message $\langle m_1, l' \rangle$ t time units ago. This follows from **Theorem 6.4** in the context of the previous delivery result involving $\mathbf{E_1}$ and the inequality $x + 2 * s_{max} * t \leq r$ that was just calculated. This result is depicted in the following diagram. The term $\langle m_1, l' \rangle_r!$ denotes an output of $\langle m_1, l' \rangle$ to distance r by $\mathbf{E_1}$; it is "remembered" by the *delivered* history relation. Similarly, $\langle m_1, l' \rangle?$ denotes an input of the same by $\mathbf{E_2}$, corresponding to the *received* relation. Both of these events are shown to have happened simultaneously t time units in the past.



9. It is now desirable to show that the entity $\mathbf{E_2}$ reacted to the message that it received. In order to do this, the following bound must be shown:

$$|l_2 - l'| - s_{max} * msgLatency + s_{max} * t < D_{pi}(m_1, m_2)$$

To this end, it is time for some more inequalities manipulation. We already know that the distance between l_1 and l' is at most s_{max} *($msgLatency$ + t). And so by the triangle inequality: $|l_2 - l'| \leq x + s_{max} * (msgLatency + t)$ Since $x < minDistComp(m_1, m_2)$ then we can deduce:

$$|l_2 - l'| < minDistComp(m_1, m_2) + s_{max} * msgLatency + s_{max} * t$$

Rearranging:

$$|l_2 - l'| - s_{max} * msgLatency < minDistComp(m_1, m_2) + s_{max} * t$$

Adding s_{max} *t to both sides:

$$|l_2 - l'| - s_{max} * msgLatency + s_{max} * t < minDistComp(m_1, m_2) + 2 * s_{max} * t$$

Reusing a result from a previous step:

$$2 * s_{max} * t \leq 2 * s_{max} * (max(adaptNotif, trans) + period(m_1) + trans(m_1))$$

and so

$$|l_2 - l'| - s_{max} * msgLatency + s_{max} * t < minDistComp(m_1, m_2) + \tag{6.3}$$

$$2 * s_{max} * (max(adaptNotif, trans) + period(m_1) + trans(varModeIndexed1)) \tag{6.4}$$

But the right hand side of this is just $D_{pi}(m_1, m_2)$. Tidying up then:

$$|l_2 - l'| - s_{max} * msgLatency + s_{max} * t < D_{pi}(m_1, m_2)$$

which is the desired condition for the reaction of the entity $\mathbf{E_2}$.

10. Now, since t is in I_{pd}, then $0 < t$. There is now enough information collected along the way during this proof to apply **Theorem 6.2** and deduce that $\mathbf{E_2}$ is $tfs(m,t)$. And so by **Theorem 6.7**, $t \leq trans(m) \leq trans$. But it has already been shown that t is in the I_{pd} of m_1 so $max(adaptNotif, trans) < t$, and so $trans < t$. But $t \leq trans$. Hence we have a contradiction and can reject the hypothesis, namely that there was an unsafe reachable state, concluding that all such states are, to the contrary, safe.

□

6.3.1 Main Supporting Results

This section contains what are believed to be the most important and/or interesting results supporting the proof of safety. It is probable that some lower level results are more interesting than higher level ones; the lower level results follow these ones in Section 6.3.2. This cannot be avoided, lest the top-down approach be abandoned. Rather, the reader may just be aware that the classification here is not absolute, and Section 6.3.2 may contain some results worth perusing. Nonetheless, higher level theorems generally tend to be of greater significance or interest to those further down the graph.

Remark 6.15 (Some Terminology)**.** The term "I.H." is used for "inductive hypothesis". ▲

Theorem 6.2 (incomp_tfs)**.** *If an entity has been in the current mode m since time t, and it receives a message with mode m' and position l_0 some non-zero t' time units ago, with $t' < t$, and the following inequality holds, with l being the entity's current position: $(|l - l_0| - s_{max} * msgLatency) + s_{max} * t' < D_{pi}(m_1, m_2)$, then it is $tfs(m, t')$.*

Proof of incomp_tfs. We proceed by induction on the proof of received. The base case, with $t' = 0$, fails, because t' is assumed positive. The discrete inductive case carries over easily. In the timed case, we first show that *currSince* carries to the previous state. This follows from the fact that, if the previous state were not *currSince*, then the only constructor of *currSince* applicable would be the discrete one, which would contradict the delay transition. We then split our reasoning into two cases: In the first case, the parameter to received in the previous state is non-zero, in the other case it is 0. For the former case, we must look at the old position, call it l_1. Now it follows from the semantics that the difference in old and new position is bounded as per the following: $|l_1 - l| \leq s_{max} * d$. Then from the triangle inequality we have that $|l_1 - l_0| \leq |l_1 - l| + |l - l_0|$. But from an assumption we have that

$$|l - l_0| - s_{max} * msgLatency + s_{max} * (t' + d) < D_{pi}(m, m')$$

Adding this to our position bound result we get

$$|l_1 - l| + |l - l_0| - s_{max} * msgLatency + s_{max} * (t' + d) < D_{pi}(m, m') + s_{max} * d$$

A bit of algebra allows us to cancel the s_{max} *d on both sides, giving

$$|l_1 - l| + |l - l_0| - s_{max} * msgLatency + s_{max} * t' < D_{pi}(m, m')$$

Then adding $-s_{max} * msgLatency + s_{max} * t'$ to the triangle inequality result we get

$$|l_1 - l_0| - s_{max} * msgLatency + s_{max} * t' \leq |l_1 - l| + |l - l_0| - s_{max} * msgLatency + s_{max} * t'$$

By transitivity then we have

$$|l_1 - l_0| - s_{max} * msgLatency + s_{max} * t' < D_{pi}(m, m')$$

This allows us to apply the I.H. yielding $tfs(m,t,i,\mathbf{N},p)$, which by application of the tfs delay constructor gives the required $tfs(m,(t'+d),i,\mathbf{N},p)$.

We are then left with the case where $t' = 0$. Now, by the same argument as before involving the triangle inequality, we have that

$$|l_1 - l_0| - s_{max} * msgLatency + s_{max} * t' < D_{pi}(m, m')$$

and so with $t' = 0$ we have

$$|l_1 - l_0| - s_{max} * msgLatency < D_{pi}(m, m')$$

Also we have already shown *currSince* holds for the previous state. From here we can apply **Theorem 6.6** to get *incoming* ($\langle m', l' \rangle$, *i*, **N**) $\vee$ $msgBadPath(i,\mathbf{N}) \vee tfs(m,0,i,\mathbf{N},p)$. The first two disjuncts fail because in both cases a discrete action is possible, and so by maximal progress it can be shown that delay is impossible in these cases. To elaborate on that, in the first disjunct, a synchronisation from interface to software process is possible; in the second disjunct, it can be shown elsewhere that the *msgBadPath* is always capable of discrete action. Which leaves us with $tfs(m,0,i,\mathbf{N},p)$, and we can proceed immediately from this to our goal by one application of the tfs delay constructor. □

Theorem 6.3 (pre_del_suff)**.** *If an entity in position l is currSince*(m,t)*, then it has delivered a message* $\langle m, l_0 \rangle$ *to a sufficient radius r and this happened t′ time units ago, with* $t' < t$ *in the pre-delivery interval of m and the distance between the sent position and the current position being no more than* s_{max} **(msgLatency + t′).*

Proof of pre_del_suff. Let us first recall that the definition of I_{pd} for mode m is ($max(adaptNotif,trans)$, $max(adaptNotif,trans) + period(m) + trans(varMode)$]). Let u = $msgLatency + max(adaptNotif,trans)$ + $period(m)$ + $trans(varMode)$. Then we can do a case analysis on $t \leq u$ or $u < t$.

Case $t \leq u$: Well, we can show by **Theorem 6.9** that there is a sufficient message delivered at $t' = t - msgLatency$ and some position l_0 is sent in that message such that $|l - l_0| \leq s_{max} * (msgLatency + t')$. By the positivity of *msgLatency*, it is easy to show $t' < t$. So it remains to show that t': $I_{pd}(m)$. Well, since by **Theorem 6.8** we have that $msgLatency + max(adaptNotif,trans) < t$, then $max(adaptNotif,trans) < t - msgLatency = t'$, which is the lower bound done. Also, since $t \leq u$ =$msgLatency$ + $max(adaptNotif,trans)$+$period(m)$ + $trans(varMode)$, then $t - msgLatency \leq max(adaptNotif,trans) + period(m) + trans(varMode)$, which is our upper bound done. Case closed.

Case $u < t$: Here we can easily show $msgLatency + max(adaptNotif,trans) + trans(varMode) < t$ by transitivity via u, so by **Theorem 6.12** we have that there was a sent message $\langle m, l_0 \rangle$ at a time t' within the window ($msgLatency$ + $max(adaptNotif,trans)$ + $trans(varMode)$, $msgLatency$ + $max(adaptNotif,trans)$ + $period(m) + trans(varMode)$]) and $t' \leq t$. By **Lemma 6.13** we can establish that $|l - l_0| \leq s_{max} * t'$.

Then by **Theorem 6.10** we know that this message was delivered to some radius r at time $t'' = t' - msgLatency$ and so within the window ($max(adaptNotif,trans) + trans(varMode)$, $max(adaptNotif,trans)$ + $period(m) + trans(varMode)$]) i.e. the sent window minus the message latency. It is clear then that t'' is also within the window $I_{pd}(m)$, since the window mentioned above is contained within $I_{pd}(m)$. Now, since from above $t' \leq t$ and $t'' = t' - msgLatency$, then by transitivity and the positivity of *msgLatency* it follows that $t'' < t$. Also, rearranging the definition of t'' we get t' = $msgLatency$ + t''. Substituting into the earlier result of bounded position we get $|l - l_0| \leq s_{max} * (msgLatency + t'')$.

We have now shown every sub-goal except the sufficiency of the delivered message. It can be shown that the coverage r must be sufficient for the mode m by assuming otherwise and proving a contradiction. Well, because

of the interval in which t'' has been shown to reside, it must be that $adaptNotif < t''$, and we have already shown that $t'' < t$, so by **Theorem 6.11**, the entity in question can be shown to be $tfs\ m\ (t'' - adaptNotif)$. Then by **Theorem 6.7**, this would imply $t'' - adaptNotif \leq trans(varMode)$. However, by the enclosing interval of t'' we also have that $max(adaptNotif, trans) + trans(varMode) < t''$, so $adaptNotif + trans(varMode) < t''$ and so $trans(varMode) < t'' - adaptNotif$. Hence a contradiction is reached and it must be accepted then that the broadcast range was in fact sufficient. This concludes the proof of all the sub-goals. □

Theorem 6.4 (delivered_received)**.** *Let's say two entities* $\mathbf{E_1}$ *and* $\mathbf{E_2}$ *are separated by a distance x. If* $\mathbf{E_1}$ *has delivered message* $\vec{v}$ *to radius r t time units ago, and* $x + 2 * s_{max} * t \leq r$*, then* $\mathbf{E_2}$ *has received* $\vec{v}$ *t time units ago. Intuitively, this makes sense. Since* s_{max} *is the maximum speed of an entity then 2*s*$_{max}$ *is the maximum relative speed between entities, granted they are not travelling near light speed of course. So if two entities are now separated by x, then t time units ago they can be separated by at most x + 2*s*$_{max}$ **t. If this latter term is then less than or equal to r, the radius of delivery, then the message sent by one entity must be received by the other, since it is within range. This is a direct feature of the semantics of this model.*

Proof of delivered_received. Induction on the proof of delivered. In the base case, we have that $\mathbf{N} \xrightarrow{\langle \vec{v} \rangle^{l}_{ri}!} \mathbf{N}'$ for some position l and $t = 0$. So our last hypothesis becomes $x \leq r$. By analysing the transition from $\mathbf{N}$ to $\mathbf{N}'$, we can see from the semantics that this can only happen when all entities within range of entity i input the message, and all others ignore. By $x \leq r$, $\mathbf{E_2}$ is within range. Therefore it inputs, which satisfies the base case of received with time parameter 0, as required. Let's now focus on the discrete case of induction. It follows from the semantics that the position of an entity doesn't change across a discrete transition, so the distance x separating the entities doesn't change between $\mathbf{N}$ and $\mathbf{N}'$. Thus by the I.H. we can show $received(\vec{v}, t, i', \mathbf{N}, p)$, and then by the discrete constructor of received, $received(\vec{v}, t, i', \mathbf{N}', p')$, where $p'$is the appropriately updated reachability proof. In the timed case, we have $delivered(\vec{v}, r, t, i, \mathbf{N}, p)$, $delivered(\vec{v}, r, (t+d), i, \mathbf{N}', p')$, $distNet(i, i', \mathbf{N}')x'$ and $x' + 2 * s_{max} * (t+d) \leq r$, the latter rearranging to $x' + 2 * s_{max} * t + 2 * s_{max} * d \leq r$. Here $distNet(i, i', \mathbf{N})d$ means that there are two entities at indices i and i' respectively in the network $\mathbf{N}$ separated in space by the distance d. But we can also prove that $x \leq x' + 2 * s_{max} * d$, where dist *i i'* $\mathbf{N}$ *x*: this follows from adding the bounds on the individual movement of entities as encoded by the semantics. Strategically adding 2*s_{max} *t to each side of the inequality we get $x + 2 * s_{max} * t \leq x' + 2 * s_{max} * t + 2 * s_{max} * d$. This latter terms has already been shown to be less than or equal r. So by transitivity $x + 2 * s_{max} * t \leq r$. There now exists enough information to show that $received(\vec{v}, t, i', \mathbf{N}, p)$ by the I.H. By the delay constructor of received then, we get $received(\vec{v}, (t+d), i', \mathbf{N}, p')$. □

Theorem 6.5 (nonFS_currSince)**.** *Let's say in a reachable network, an entity is in non-fail-safe mode m. Then that entity has been currSince*(m, t) *for some time t.*

Proof of nonFS_currSince. Induction on the proof of reachability of $\mathbf{N}$. In the base case, $\mathbf{N}$ is initial. Hence all its constituent entities are in fail-safe modes. So the assumption that entity i is in mode m which is not fail-safe is contradictory, and the case can be immediately dismissed. Which leaves the inductive cases. Here, we proceed via a case analysis on whether the entity in the previous state was $currMode(m)$. If it was, then by the inductive hypothesis, we get $currSince(m, t)$ for the previous state which immediately follows on to this one by the inductive definition of *currSince*. Otherwise, since an entity will always have some current mode, the entity in the previous state must have been $currMode(m)$' for some $m' \neq m$. This implies that the current mode has changed between states. By **Lemma 6.14** this can only happen when the entity performs a τ (internal) transition, with the software component contributing an output on mCurr and the mode-state contributing an input. And so one of the sub-components $\mathbf{P}_1$, $\mathbf{P}_2$ or $\mathbf{P}_3$ must have performed this output. We eliminate the possibilities $\mathbf{P}_1$ and $\mathbf{P}_3$ by an analysis of the protocol components, noticing that neither the listening component nor the broadcast component can ever output on mCurr. Thus, the output must have come from $\mathbf{P}_2$, the overlap component. By analysis of the possible overlap states it is possible to deduce that to output on this channel, $\mathbf{P}_2$ must have been in the state switchCurrState, while the next state had $\mathbf{P}_2$' in *switchListen*. There is also a tfs case, but this fails because $fs(m)$, i.e. the assumption that m is fail-safe, contradicts one of the hypotheses.

Now it can be shown that the switchCurrState can only be reached when the *nextSince* relation holds, for some t i.e. $nextSince(m,t,i,\mathbf{N},p)$. Then by the base case of *currSince*, it can be shown that $currSince(m,t,i,\mathbf{N},p)$ holds. □

Theorem 6.6 (instant_incomp_tfs)**.** *A precursor to incomp_bad, this says that if a message was just received this instant and it's possibly incompatible with the current mode and position, then either a) it is pending consumption, b) the message in question is being processed along the msgBadPath or c) the receiving entity has just initiated a transition to tfs this instant.*

Proof of instant_incomp_tfs. Induction on received. Incoming is easy to show for the base case- it follows from the network semantics. The delay inductive case can be thrown out because the time parameter to received is 0. So all that remains is the discrete inductive case. Now, since received and *currSince* share the same history, it can be shown that $currSince(m,t,i,\mathbf{N},p)$ must hold for the previous state or $nextSince(m,t,i,\mathbf{N},p)$ must hold.

If *nextSince* holds in the previous case, with *currSince* holding now, then it can be shown that the action linking the states was a τ leaving the interface unchanged. An application of **Lemma 6.15** yields *incoming* $(\langle m',l'\rangle,\ i,\ \mathbf{N}) \vee msgAbortPath(i,\mathbf{N})$. We proceed by analysing the two disjuncts separately. The case of *incoming* is easy: we have already shown that the interface is preserved and so this relation is also preserved. The case of *msgAbortPath* yields a contradiction. We can show that since a switch from *nextSince* to *currSince* occurred for the entity in question, then it is in a switchState. From here, we can show that the listener component must be paused, a fact evident on inspection of the protocol processes. But paused is disjoint from a msgAbortState by definition, hence the contradiction.

For the case in which the previous state was *currSince*, it is possible to apply the I.H. Also, it is clear that *currSince* implies $currMode(m)$. Now, if the previous state was *incoming*, then by **Lemma 6.16** one of the first two disjuncts holds, and the proof is done. On the other hand, if *msgBadPath* holds, it can be shown elsewhere that it is either preserved or goes to $tfs(m,0,i,\mathbf{N},p)$, which in turn is preserved with time parameter 0. Hence the disjunction is always satisfied. The preservation of $tfs(m,0,i,\mathbf{N},p)$ relies on the fact that such a tfs with 0 time parameter must be either tfsStart, tfsNext or a guarded tfsCurr, the latter being unable to leave the state until delay passes. This can be seen by examining the overlap component of the protocol. □

Theorem 6.7 (tfs_bound)**.** *If an entity is tfs m for a time of t, then this t is bounded from above by $trans(varMode)$. This bound is essential to the correctness of the protocol: it guarantees that all entities will reach a fail-safe mode within some given time once they have started a transition to such a mode. This is important because without a guarantee of timely reaction, a beaconing entity would never be able to assert in due time that its neighbourhood is safe and so would never be able to enter any non-fail-safe mode without risking an incompatibility.*

Proof of tfs_bound. This proof proceeds via a case analysis of the tfsStates. It is easy to show that no time can pass for tfsListen, tfsBc, tfsNext and tfsStart, because they are all capable of some discrete action or another. All that remains then is to prove a special result, namely **Lemma 6.17** for tfsCurr, which yields the bound. □

Theorem 6.8 (currSince_lower_bound)**.** *Assume it can be shown that a network is $currSince(m,t,i,\mathbf{N},p)$, then it is possible to show that the time parameter to this relation is always greater than*

$$msgLatency + max(adaptNotif, trans)$$

which is the lower bound for the pre-sent interval. This property is important because, combined with properties relating message broadcast to currSince, it entails that an entity has given its environment sufficient warning of it current mode status i.e. enough time has passed for a message sent by the entity to propagate to the environment, and for all receiving entities to react to this message if they calculate that reaction is necessary.

Proof of currSince_lower_bound. By induction on *currSince*. Oddly, the inductive cases are the easy ones here- because the parameter t to *currSince* is non-decreasing over these. It's the base case that requires a little more

mental strain. Well, we notice that the base case of *currSince* involves a proof of *nextSince* for the previous state. It also enforces the entity in the previous state to be switchCurrState m. Now, applying **Lemma 6.18** yields that *nextSince*, in this previous state, implies the lower bound, and since t is the same in the base case from premise to conclusion (passed on like a relay bat), this lower bound carries on the current state and *currSince*, as required. It is worth noting that most of the "work" of this theorem is done in the analogous theorem involving *nextSince*, which is then "lifted" to here giving the bound for *currSince*. □

Theorem 6.9 (currSince_fst_deliv_suff)**.** *If an entity in position l is currSince(m,t), then it has delivered a message $\langle m, l_0 \rangle$ to a sufficient radius r and this happened $t - msgLatency$ time units ago. Also, l_0 differs from l by at most s_{max} *t.*

Proof of currSince_fst_deliv_suff. Induction on *currSince*. In the base case, the previous entity was $nextSince(m,t)$ and in switchCurrState. Then **Lemma 6.18** gives $msgLatency + max(adaptNotif, trans) < t$. This is easily weakened to $msgLatency + adaptNotif < t$. Also, it is clear from the discrete nature of the transition in this case that the position is preserved. Hence **Lemma 6.19** yields $delivered(\langle m, l_0 \rangle, r, (t - msgLatency), i, \mathbf{N}, p) \wedge suff(m,d) \wedge |l - l_0| \leq s_{max} * t$ for some r and l_0. This closely resembles the goal, except with $delivered(\langle m, l_0 \rangle, r, (t - msgLatency), i, \mathbf{N}, p)$ instead of $delivered(\langle m, l_0 \rangle, r, (t - msgLatency), i, \mathbf{N}', p')$. This last touch is shown by the application of the discrete delivered constructor, and the case is done. The discrete inductive case is almost identical: backwards preservation of position, apply I.H., apply discrete constructor on delivered. The delay case is a little more complicated. It follows from the semantics that the entity in the previous state was in some position l_1 such that $|l - l_1| \leq s_{max} * d$. From here we apply the inductive hypothesis. The proof of delivered carries over by the delay delivered constructor. The sufficiency of the message is preserved. The bound on the distance then is shown by the triangle inequality. Elaborating on this last point, we want to show $|l - l_0| \leq s_{max} * (t + d)$. We have both that $|l - l_1| \leq s_{max} * d$ and $|l_1 - l_0| \leq s_{max} * t$. The triangle inequality also gives us that $|l - l_0| \leq |l - l_1| + |l_1 - l_0|$. So the goal is obtained by adding the inequalities and employing transitivity. □

Theorem 6.10 (sent_delivered)**.** *If an entity has sent a message $\vec{v}$ t time units ago, and that time is greater than the message latency, then said message was delivered to some radius, and the time since delivery is less than the time since sending by the amount message latency.*

This property is intuitive when one considers the semantics of the entity and interface languages and the definitions of sent and delivered. On sending a message, clearly that message gets buffered in the output queue of the interface, with timestamp msgLatency. Now, since msgLatency $< t$, at some point since the message was sent, the time msgLatency is reached, and the timestamp on the outgoing message reaches zero. From here, the semantics dictate that such a message must be broadcast before delay is possible, and the base case for delivered is satisfied. Then the remaining time passes, which is $t - msgLatency$, and the time parameter to delivered is updated to such, as required. Note that there are no constraints on the radius here. This agrees with the rules for message delivery as per the operational semantics of the network calculus, in which an arbitrary radius may be chosen for message delivery in order to model arbitrary variations in coverage.

Proof of sent_delivered. Induction on *sent*. The base case can be immediately discarded due to the contradiction of $t = 0$ and $msgLatency < t$, given that *msgLatency* is positive. In the discrete inductive case, the time parameter does not change since the previous state, and so the inductive hypothesis yields $delivered(\vec{v}, r, t\text{-}msgLatency, i, \mathbf{N}, .)$ Then by the discrete constructor of *delivered* we can carry the result over to this state i.e. $delivered(\vec{v}, r, t\text{-}\ msgLatency\ , i, \mathbf{N}', .)$ For the timed case, there are two sub-cases. If $msgLatency < t$, then the goal is achieved via the inductive hypothesis and timed constructor of delivered. Else, $t \leq msgLatency$. This is split again into $t < msgLatency$ and $t = msgLatency$. The case $t < msgLatency$ gives a contradiction. We achieve the contradiction by first applying **Lemma 6.20** to get $outgoing\ (\vec{v}, (msgLatency - t), i, \mathbf{N})$. Then we apply **Lemma 6.25** to get $d \leq msgLatency - t$ and so $t + d \leq msgLatency$. But from our hypothesis we have $msgLatency < t + d$, so we arrive at a contradiction. Thus we conclude $t = msgLatency$, which is the exact value of the timestamp of any message upon entering the output queue. Well, then, it can easily be shown

that after such a delay, either the message has timed out and is pending output, or it has just been output this instant i.e. $outgoing$ ($\vec{v}$, 0, i, $\mathbf{N}$) or there exists r such that $delivered(\vec{v}, r, 0, i, \mathbf{N}, .)$ The LHS fails, again by contradiction via a corollary of **Lemma 6.25**, which says that t = 0 implies no delay is possible. Therefore we conclude that $delivered(\vec{v}, r, 0, i, \mathbf{N}, f)$or some r, and then by the delay constructor of delivered we have our goal of $delivered(\vec{v}, r$,0 - d $, i, \mathbf{N}', .)$ □

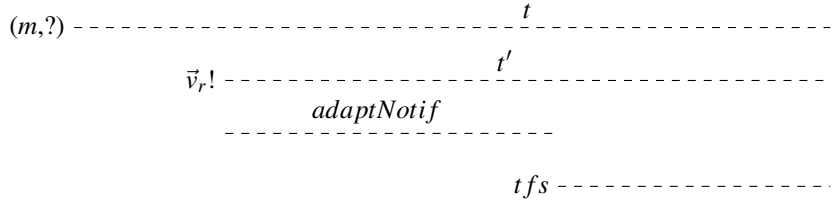
Theorem 6.11 (insuff_react)**.** *If an insufficient message was delivered t' time units ago, with $adaptNotif < t'$, and the sender has been currSince for some time t greater than t', then we are tfs for (t'- $adaptNotif$) time units. This theorem essentially states that degradation in coverage is always followed by a reaction of the entity in question transitioning to a fail-safe mode. The contrapositive of this is that, if no fail-safe reaction has been initiated, then all previous messages since entering this mode must be sufficient.*

Proof of insuff_react. Induction on *currSince*. In the base case, the previous state was *nextSince*. From here, the proof proceeds by contradiction. First, we observe that no time has passed since the previous state. Then it can be shown to follow from the semantics that the action linking the previous state to the current one must be a τ. These assertions then imply delivered must be true for the previous state also, because the base case of delivered doesn't match against a τ. But then **Lemma 6.22** gives $suff(m, r)$, in direct contradiction with a hypothesis.

The discrete inductive case follows straightforwardly from an application of the I.H. and the discrete constructor for tfs.

In the delay inductive case, the proof proceeds by case analysis on whether $adaptNotif < t'$ or not. First, let's assume it is. We already have $t' < t$ from $t' + d < t + d$, so then we can apply the I.H. and the delay constructor of tfs and the case is done. For the case of $t' \leq adaptNotif$ we have either $t' < adaptNotif$ or $t' = adaptNotif$. Using a combination of **Lemma 6.28** for the $t' < adaptNotif$ case and **Lemma 6.29** for the t' = $adaptNotif$ case then we can show $incomingNotif(\langle r, m, l\rangle, 0, i, \mathbf{N}) \vee notifBadPath(i, \mathbf{N}) \vee tfs\ m\ 0i\mathbf{N}$. The first two disjuncts fail because delay can be shown to be impossible in these cases. And so we are left with $t' = adaptNotif$ and $tfs(m, 0, i, \mathbf{N}, p)$. In this case, we apply the tfs delay constructor to show $tfs(m, d, i, \mathbf{N}, p)$, which is exactly our goal simplified from $tfs(m, (adaptNotif + d - adaptNotif), i, \mathbf{N}, p)$.

The following picture roughly captures the intuition behind this result. This deviates slightly from the conventions of earlier diagrams in that all time lines refer to the same history just one entity; they are merely broken up for readability. The first, second and fourth of the time lines depict events, the other is merely to show the temporal relationship between the second and fourth. The diagram can be interpreted as saying that if an insufficient delivery happens, according to the second time line, and then $adaptNotif$ time units are allowed to pass as per the third time line, then the tfs relation can be seen to hold since the $adaptNotif$ units have elapsed, as represented by the fourth time line. The first time line stipulates that this all happens within the context of the entity being *currSince* for longer than the time since delivery.

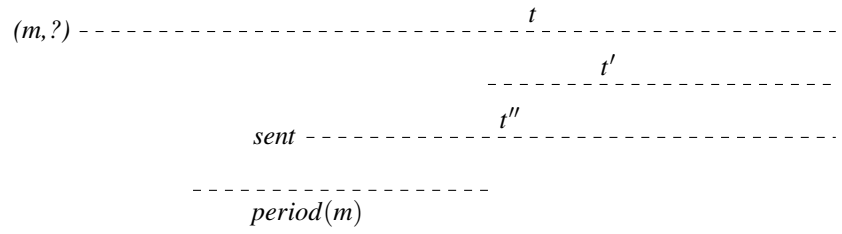


□

Theorem 6.12 (currSince_periodic_sent)**.** *Let's say an entity has been broadcasting in the current mode m since t time units ago. Then for any $t' < t$, we can assert that a message was sent containing the mode in question m and some position l_0 within the window t' and t' + $period(m)$. We also know that the message was sent at most t units ago.*

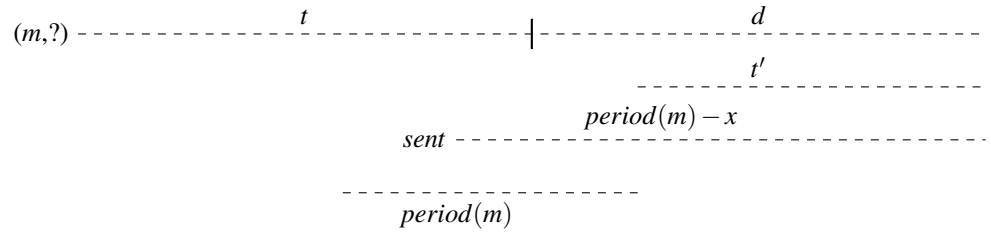
This property asserts that it is possible to choose any time since the last mode transition was initiated and find a broadcast in the locality of that time at most period time units away. Equivalently, this means a periodic sequence of broadcasts has happened since the last mode switch. Note that this sequence need not be uniformly periodic, with messages being separated by an exact period each time, rather the time separating consecutive

messages is bounded *by the amount period*(m)*, which suffices in the context of the overall proof. The following diagram is similar to that of* ***Theorem 6.11****, depicting the theorem statement via various past events and their temporal relationships. The third time line represents the sent message while the final one depicts the interval of time in which the message is hypothesised to have been sent.*



Proof of currSince_periodic_sent. Induction on *currSince*. In the base case, we have that the previous state was *nextSince*. In which case we apply **Lemma 6.23** and the discrete *sent* constructor to obtain the goal. The discrete inductive case is easy: apply I.H. and then the discrete constructor for *sent*. Which leaves the delay case.

Well, here we do a case analysis on $d \leq t'$ or $t' < d$. In the former case, we know then that $t' = t'' + d$ for some time t''. Now, by one of the hypotheses we have that $t' < t + d$, so $t'' < t$, and so we can apply the I.H. in the context of t'' and the previous state and from there an application of the *sent* delay constructor concludes the sub-goal. Then all that remains is the delay case where $t' < d$. Well we know that the previous state was $bcWait(m, x+d)$, because *bcReady* is urgent and cannot delay, and so by **Lemma 6.24** we get $sent(\langle m,l\rangle, period(m) - (x + d))$ for the previous case. Then by the delay constructor of *sent*, we get $sent(\langle m,l\rangle, period(m) - x)$ for the current state. The following diagram, in a similar vein to those preceding it, attempts to shed some light on the delay case for this proof where $t' < d$. Note that the time line for *currSince* is split by a vertical bar into two segments of lengths t and d respectively; its entire length is the sum $t + d$.



But since $t' < d < period(m) - x$, then $t' < period(m) - x$. So the lower bound of the required interval holds. But clearly it is also the case that $period(m) - x \leq t' + period(m)$, and so the upper bound holds. Therefore the message resides in the correct interval. □

6.3.2 Lower Level Results

The results put forth here are, for the most part, less interesting than those covered thus far. As a general rule, then, they should be given less attention than the others. The level of detail given in the proofs may seem, in many cases, superfluous; if this is so then the reader is encouraged to skip such material. Still, the proofs are included so that the interested reader may peruse them to a desired level of detail. After all, there is no definitive way of classifying a result as "interesting" or not; it may even be the case that a few of these results could be considered more interesting or important than the higher level results of Section 6.3.1

Lemma 6.13 (sent_pos_bound)**.** *If a message* $\langle m, l_0\rangle$ *was sent t time units ago by some entity, then the distance between* l_0 *and the position of that entity is at most* s_{max} **t. To understand this result, it is best to first examine the base case for sent. In this case, the software component has just sent the message in question to the interface. Now, a quick analysis of the protocol process reveals that the only transitions capable of sending a message to the interface are immediately preceded by an input of the current position. Furthermore, the state waiting to send is urgent i.e. no time can pass while in it and hence the position remains the same. Hence, the sent position is always the current position to begin with. The remaining cases follow by induction, as per the*

following proof, and so are not covered here. This theorem is essential in that it links the content of a message to the actual position of the sender. This link carries forward to delivered messages (as opposed to sent ones) i.e. ones that reach other entities. From there, other entities can then deduce, by looking at the contents of a message, a bound on the position of the entity that sent the message. This information then allows them to adapt their behaviour accordingly.

Proof of sent_pos_bound. By induction on the proof of *sent*. The base case would give us that the network **N** is reachable. From this we could deduce from an important general result that the software component of the entity i is $\mathbf{P}_1 \mid \mathbf{P}_2 \mid \mathbf{P}_3$, a parallel composition of a broadcast, overlap and listener process respectively. We could then show via another general result that one of the sub-components must have been responsible for the output of $\vec{v}$ on the *chanOutP* channel. By analysis of the protocol components, we would then show that there are only two cases that fit, one for $\mathbf{P}_1$ the broadcast process and one for $\mathbf{P}_2$ the overlap process. We can also show that the states which perform the outputs in question are ready to output $\langle m, l_0 \rangle$ with $l_0 = l$ = position of sender. Then the goal is reduced to: $|l - l| \leq s_{max} * 0$, simplifying to $0 \leq 0$, which is trivial.

The inductive cases would follow immediately from the definition of *sent* and the semantics. If a discrete action happens, in which case both t and the position of entity i don't change, the goal carries over directly from the previous case. Alternatively, a delay happens, in which case the semantics dictate that the new position l' of entity i differs from its old position by at most $s_{max} * d$. And so by the triangle inequality the difference between l' and the sent position l_0 is at most $s_{max} * t + s_{max} * d = s_{max} * (t + d)$, which is exactly the goal because the time parameter of *sent* for the current state is $(t + d)$. □

Lemma 6.14 (curr_switch_proc)**.** *If the current mode has changed across a τ transition, then the software process outputs on mCurr, and the mode state inputs on the same channel. This follows directly from the semantics.*

Proof of curr_switch_proc. By inversion on the discrete entity transition. We can eliminate all cases where the mode state is preserved or the action is not τ, leaving the case corresponding to our goal. □

Lemma 6.15 (instant_incomp_abort)**.** *This says that if a message was just received this instant and it's possibly incompatible with the pending mode and position, then either a) it is pending consumption or b) the message in question is being processed along the msgAbortPath.*

Proof of instant_incomp_abort. The proof of this theorem is analogous in structure to **Lemma 6.21**. The overall strategy is an induction on the delivered relation and use of some auxiliary results. □

Lemma 6.16 (incomp_incomingNet_bad)**.** *If an incoming message is incompatible with the current mode and position, then in the next state after a discrete transition, it is either still incoming or the entity in question has entered the msgBadPath. This result is used to support a higher level result dealing with the reaction of entities to incoming messages from other possibly incompatible entities.*

Proof of incomp_incomingNet_bad. We case analyse the action of the discrete transition. For the cases where a is either input, output or ignore, we can show that incomingNet is preserved, satisfying the LHS. In the remaining case, a is a τ action. Then we can show that either incomingNet$\vec{v}0i\mathbf{N}'$ or the τ is the result of an output of $\vec{v}$ by the interface and an input of the same by the software process. Then by the base case of *msgBadPath*, it can be shown that $msgBadPath(i, \mathbf{N}')$, which is the RHS of the goal. □

Lemma 6.17 (tfs_transGuard)**.** *If an entity is $tfs\ t$ in the state $tfsCurr$, then there is some t' such that the mode transition guard for that entity is t' and the sum of t and t' is at most $trans(varMode)$. This result is essential in establishing the upper bound on tfs, a higher level result which immediately follows when it is observed that t', being a time, must be at least 0. In other words, that bound is a weakening of this theorem.*

Proof of tfs_transGuard. Induction on tfs. The base case fail because it implies the entity is in tfsStartState, contradicting the hypothesis that it is in $tfsCurr$.

In the discrete inductive case, we perform a case analysis on whether or not the previous state was $tfsCurr$. If it was, then application of the I.H. is possible. Then we assert that the mode state and hence the transGuard doesn't change across the transition. This follows from the general observation that only the overlap component can change the mode state and so, since the overlap component has remained stagnantly in $tfsCurr$, it could not have done so. However, if the entity in the previous state wasn't in $tfsCurr$, then it is evident upon inspection of the overlap component that it was in the tfsNextState and that the transition linking the states was an output to the mode state of the next mode. But then by the mode state semantics, the guard on the mode state on entering $tfsCurr$ would be $tt(m_0,m)$, which by a system assumption is less than or equal $trans(varMode)$. Also, it can be shown that all states up to this point since entering tfs are urgent, and so the time parameter to tfs is 0 on entering this state. So the bound holds.

Now, for the delay inductive case, we know that delay preserves $tfsCurr$, so we can apply I.H. to the previous state. Then we know that the transGuard decreases and the t from tfs increases proportionately, and so the sum remains constant. □

Lemma 6.18 (nextSince_switchCurr_lower†)**.** *If an entity is $nextSince(m,t)$, and the same entity is in the state $switchCurr(m)$, then we can show that $msgLatency + max(adaptNotif, trans) < t$. This result precedes the lower bound result for currSince, the latter following in a "relay" fashion. A quick intuition as to why this result holds can be gained by observing the overlap component of the protocol, upon which switchCurr is based, and noticing that the transition leaving the wait state is guarded by a time greater than the amount mentioned here. Now, this transition must be taken before entering the switchCurr, and the nextSince relation records the time that passes prior to taking this transition. Thus, by the time this state is reached, the lower bound can be seen to hold.*

Proof of nextSince_switchCurr_lower. The proof combines induction on *nextSince* and case analysis on *switchCurr*. The base case fails because it implies that the state is *ovWait*, contradicting the hypothesis that the entity is in *switchCurr*. For the both inductive cases, we case analyse whether the previous state was *switchCurr*. If this was true, then by the I.H. the t parameter in that state exceeds the lower bound, and so, by monotonicity of *nextSince* in both delay and discrete cases, so will the t' in this state. Otherwise, the previous state is not $switchCurr(m)$. In which case by inspection of the overlap component it can be seen that it must be $switchBc(m)$, and the transition must be discrete of course. We then prove elsewhere a similar result to this one for *switchBc* and since the t parameter is the same from the last state to this one because the transition is discrete, that result carries forward. □

Lemma 6.19 (nextSince_fst_deliv_suff)**.** *If an entity has been awaiting a transition to a mode m for the past t time units, and this t is bounded from below by $msgLatency + adaptNotif$, then the entity in question has delivered a message of $\langle m, l_0 \rangle$ to a sufficient radius r and l_0 differs from the current position by at most s_{max} *t. Establishing this initial sufficient message delivery is essential: it means that on entering a non-fail-safe mode, an entity is guaranteed to have notified a sufficient neighbourhood of entities of its whereabouts and its mode, allowing them to react accordingly if they need to to ensure safety.*

Proof of nextSince_fst_deliv_suff. Well since $msgLatency + adaptNotif < t$, then $msgLatency < t$. Now, since a message is sent immediately upon entering next since, and by **Theorem 6.10** we have that $delivered(\langle m, l_0 \rangle, r, t\text{-}msgLatency, i, \mathbf{N}, p) \wedge |l - l_0| \leq s_{max} * t$. So all that remains to be proven is the sufficiency of the radius r for the mode m. Now, again taking our inequality $msgLatency + adaptNotif < t$ we can easily obtain $adaptNotif < t - msgLatency$. Also due to the positivity of $msgLatency$ we have $t - msgLatency < t$. Then **Lemma 6.22** gives the remaining goal of $suff(m,r)$. □

Lemma 6.20 (sent_outgoing)**.** *If an entity has sent a message $\vec{v}$ t time units ago, and that time is less than the message latency, then said message is in the output queue of the interface component with timestamp $msgLatency - t$. This follows from the definition of sent and the semantics of the entity language.*

Proof of sent_outgoing. Induction on *sent*. In the base case we have that $\mathbf{I} \xrightarrow{chanInP\langle\vec{v}\rangle?} \mathbf{I}'$, where $\mathbf{I}$ is the interface of the entity in question. By analysing this interface transition it can be seen that $\langle\vec{v}, msgLatency\rangle$ is at the head of the output list of $\mathbf{I}'$, and so $outgoing(\vec{v}, msgLatency, i, \mathbf{N})$ holds, which is the goal exactly since $t = 0$.

In the delay case for induction, we can immediately infer from $t + d < msgLatency$ that $t < msgLatency$ and so *outgoing* ($\vec{v}$, $msgLatency - t$, i, $\mathbf{N}$). Then since $\mathbf{N} \xrightarrow{d} \mathbf{N}'$, **Lemma 6.25** yields *outgoing* ($\vec{v}$, $msgLatency - t - d$, i, $\mathbf{N}$) = *outgoing* ($\vec{v}$, ($msgLatency$- (t + d)), i, $\mathbf{N}$), which is the required goal.

In the discrete case, the I.H. can be applied since t hasn't changed; this gives *outgoing* ($\vec{v}$, $msgLatency - t$, i, $\mathbf{N}$). We then observe that since $t < msgLatency$, then $0 < msgLatency - t$, and so by **Lemma 6.32** the *outgoing* relation would be preserved from the previous to the current state, giving *outgoing* ($\vec{v}$, $msgLatency - t$, i, $\mathbf{N}'$) as required. □

Lemma 6.21 (instant_insuff_abort)**.** *If a message with mode m was delivered exactly $adaptNotif$ time units ago, and the coverage of the message was insufficient with respect to said mode, and we have been awaiting a transition to m for some t time units greater than $adaptNotif$, then one of two things holds. Either a) A notification of the message and its coverage is pending immediately i.e. with timestamp 0 or b) the entity in question is on the notification abort path. This result can be seen as a "base case" demonstrating the reaction of entities to coverage degradations. This is an essential behaviour for entities because, when the coverage does become insufficient, an entity no longer can guarantee that its neighbours have "heard" its beaconing and therefore it must conservatively adapt its behaviour through a transition to a fail-safe mode.*

Proof of instant_insuff_abort. Induction on *delivered*. The base case fails because $adaptNotif$ is positive and therefore not equal to 0.

For the discrete inductive case, we apply the I.H. to the previous state, yielding $incomingNotif(\langle r,m,l\rangle, 0, i, \mathbf{N}) \vee notifAbortPath(i, \mathbf{N})$. Now we do a case analysis on this disjunction. First off there is $incomingNotif(\langle r,m,l\rangle, 0, i, \mathbf{N})$. We can show then by **Lemma 6.26** that the goal holds in the next state.

Then there's the case of $notifAbortPath(i, \mathbf{N})$. In this case, **Lemma 6.27** can be applied, showing that either $notifAbortPath(i, \mathbf{N}')$ or $\neg nextSince(i, \mathbf{N}')$. The former is exactly the RHS of the goal. The latter causes a contradiction because $nextSince(m, t, i, \mathbf{N}', p)$ implies $nextSince(i, \mathbf{N}')$.

For the delay case, we apply **Lemma 6.28** to the previous state to obtain $incomingNotif(\langle r,m,l\rangle, (adaptNotif - (adaptNotif - d)), i, \mathbf{N}) = incomingNotif(\langle r,m,l\rangle, d, i, \mathbf{N})$. Then by **Lemma 6.30** we get $incomingNotif(\langle r,m,l\rangle, 0, i, \mathbf{N}')$, the left disjunct of the goal. □

Lemma 6.22 (deliv_next_suff)**.** *If a message was delivered t' time units ago, and we have been waiting to enter the next mode m for some time t, with $t' < t$, and the time since delivery t' is greater than $adaptNotif$, then the message delivery in question was sufficient. Well, this can be seen to follow from the semantics. Whenever a message is delivered, a corresponding notification message is buffered containing the coverage of the message with timestamp $adaptNotif$. But since t' is greater than $adaptNotif$, this timestamp must have elapsed before now. At which point the notification must have reached the software component. Now, an insufficient radius of delivery would prompt the software component to abort the current mode transition immediately. However, the current mode transition has been in place for t time units, with $t' < t$, and so this abort could not have happened. Therefore the coverage must have been sufficient. It may be worth noting that the proof of this theorem bears resemblance to that of* ***Theorem 6.2****.*

Proof of deliv_next_suff. By induction on *delivered*. The base case fails because of the contradiction implied by $adaptNotif < 0$.

For the inductive cases, we can show that since *delivered* and *nextSince* share the same history p, the previous state was also *nextSince*. In the discrete inductive case we notice that t and t' haven't changed and so we just apply the I.H. In the timed case, we do a case analysis on the time t' in the previous case. On the one hand, we may have $adaptNotif < t'$, in which case we can just apply the I.H. Which leaves us with the other case, $t' \leq adaptNotif$, the only part of the proof that requires real thinking.

Well, this requires a further case analysis on $t' < adaptNotif$ or $t' = adaptNotif$. If $t' < adaptNotif$, then by **Lemma 6.28** we get $incomingNotif(\langle r,m,l_0\rangle,(adaptNotif - t'),i,\mathbf{N})$. Now, it is already a hypothesis that $adaptNotif < t' + d$, which rearranges to $adaptNotif$- $t' < d$. However, **Lemma 6.30** gives us that $d \leq adaptNotif - t'$, which is a contradiction. So we reject this case and conclude that $t' = adaptNotif$. We proceed from here by reductio ad absurdum, that is assume $\neg suff(m,r)$ towards a contradiction. Well, there is enough information to apply **Lemma 6.21** which gives $incomingNotif(\langle m,l,r\rangle,0,i,\mathbf{N}) \vee notifAbortPath(i,\mathbf{N})$. The first disjunct gives a contradiction via an application of **Lemma 6.30** yielding $d \leq 0$, impossible for a strictly positive delay. The other disjunct fails also because it can be shown that delay is impossible whenever an entity is on the $notifAbortPath$. This gives the required contradiction and we are forced then to accept the sufficiency of the coverage. □

Lemma 6.23 (nextSince_periodic_sent)**.** *Let's say an entity has been awaiting a transition to the mode m since t time units ago. Then for any $t' < t$, we can assert that a message was sent containing m, the mode in question, and some position l_0 within the window t' and t' + $period(m)$. We also know that the message was sent at most t time units ago. We notice that this result is analogous to that of currSince_periodic_sent, the only difference being it deals with the nextSince relation rather than the currSince relation. In fact, this result is essential to proving the result for currSince.*

Proof of nextSince_periodic_sent. This proof resembles that of **Theorem 6.12**, and proceeds by induction on *nextSince*. The base case with $t = 0$ fails because then we would have the contradictory hypothesis $t' < 0$, violating the non-negativity of time. The discrete inductive case follows by simple application of the inductive hypothesis and then the discrete *sent* constructor. Which leaves just the timed inductive case. Here, we perform a case analysis of the various states for *nextSince*. It happens that the only such state capable of delay is *ovWait*. We then proceed analogously to **Theorem 6.12**. □

Lemma 6.24 (bcWait_sent)**.** *If we are in the wait state of the broadcast component, and the clock value x is non-zero, then a message containing the mode in question and some position was sent exactly ($period(m)$ - x) time units ago. On inspection of the broadcast component of the protocol, it can be seen that the wait state is only entered with a non-zero time parameter after the ready state via an output of a message to the interface. From here, the required relationship can be shown, the details of which are left to the proof below. This result is essential in establishing the periodicity of sent messages: if at all times a message was sent at most $period(m)$ time units ago, then a periodic sequence clearly ensues. Now, the sequence may not be uniformly periodic, with messages being separated by an exact period each time; rather the time separating consecutive messages is* bounded *by the amount $period(m)$, which suffices for higher level purposes.*

Proof of bcWait_sent. Induction on the proof of reachability of the network. The base case fails because we know all entities are sleeping to begin with, contradicting the hypothesis that the entity is in *bcWait*. The delay inductive case is straightforward, following from an application of I.H. and then the delay constructor for *sent*.

The discrete inductive case has two major cases. In the first case, the previous state was $bcWait(m, x)$, with the same parameters m and x as the current state, in which case the I.H. and the discrete *sent* constructor suffice. Else it can be seen that the previous state was either $bcWait(m, x)$ and $x = 0$, $sleeping$ and $x = 0$, or $bcReady(m, l)$ and the software component output $m\ l$ on $chanOutP$ and $x = period(m)$. The first two cases fail because they contradict the hypothesis of $0 < x$. So we conclude that the third case must hold. Then by the base case of *sent* we have $sent(\langle m,l\rangle,0,i,\mathbf{N},p) = sent(\langle m,l\rangle,(period(m) - period(m)),i,\mathbf{N},p)$. □

Lemma 6.25 (outgoing_del†)**.** *If $\vec{v}$ is outgoing with timestamp t, and the network can delay, then in the derivative network $\vec{v}$ is outgoing with timestamp $t - d$. A useful corollary of this is that if t = 0, then delay is impossible.*

Proof of outgoing_del. This follows from the definition of *outgoing* and the semantics of timed lists. □

Lemma 6.26 (insuff_incomingNotif_abort†)**.** *If a notification message of insufficient coverage for a next-mode message is pending with timestamp 0, and the system evolves via a discrete transition, then either the message is still pending, or the entity in question has reacted by entering the notifAbortPath. This result is a building block towards showing that insufficient message coverage is always followed by a reaction: in this case since the message corresponds to the next mode, the reaction is to initiate an abort of the mode transition.*

Proof of insuff_incomingNotif_abort. First, a case analysis is performed on the action δ. For the cases where a is either input, output or ignore, it can be shown that *incomingNotif* is preserved; the LHS of the disjunction then holds. In the remaining case, δ is a τ action. Then we can show that either $incomingNotif(\langle r,m,l\rangle,0,i,\mathbf{N}')$ or the τ is the result of an output of $\langle r,m,l\rangle$ by the interface on the channel *chanAN* and an input by the process on this channel. Then by the base case of $notifAbortPath$, we have that $notifAbortPath(i,\mathbf{N}')$, giving us the RHS of the goal. □

Lemma 6.27 (notifAbort_not_nextSince†)**.** *If an entity is on the notification abort path, and a discrete transition happens, then in the next state it is either still on the notification abort path, or it is not in a nextSince state. Following from the result* ***Lemma 6.26****, this result ensures that an insufficient message coverage in due time causes the transition to the next mode to be aborted.*

Proof of notifAbort_not_nextSince. By analysis of the transitions of the $notifAbortPath$, it can be shown that once an entity is on the $notifAbortPath$ now, it is either still on the path in the future state or the action δ is a τ caused by the entity in question, and the listener component outputs on abort as the half action of the τ. If the entity happens to be still on the path, then the goal immediately follows via the LHS of the disjunction. Otherwise, the abort output from the listener must have been matched by an input of the same by one of the other protocol components.

Now, a quick glance at the broadcast component reveals that this could not have been responsible for the input, and so the overlap component must have performed the input. We can strengthen this further by analysing the overlap component and discovering that the derivative state must be either *dormant*, *ovAbort*, $tfsCurr$ or $tfsListen$- none of which are *nextSince* states, and so follows the goal by the RHS of the disjunct. □

Lemma 6.28 (delivered_incomingNotif)**.** *If a message was delivered at a time less than $adaptNotif$ ago, then a notification of that message's delivery is enqueued with time stamp $adaptNotif - t$. This follows from the semantics of the entity language and of timed lists.*

Proof of delivered_incomingNotif. Induction on delivered. The base case enforces an output of $\vec{v}$ to r by i. By looking at the semantics, we can see that this is only possible when the notification list of entity i also buffers $\langle r,\vec{v}\rangle$ with timestamp $adaptNotif$. Hence we have $incomingNotif(\langle r,\vec{v}\rangle,adaptNotif,i,\mathbf{N})$, which is our goal because $t = 0$. In the discrete inductive case, we have by I.H. that $incomingNotif(\langle\langle r,\vec{v}\rangle\rangle,(adaptNotif - t),i,\mathbf{N})$, and since $t < adaptNotif$ then $0 < adaptNotif - t$, so by **Lemma 6.31** we get $incomingNotif(\langle r,\vec{v}\rangle,(adaptNotif - t),i,\mathbf{N}')$. Which just leaves the delay inductive case. Well, if $t + d < adaptNotif$, then $t < adaptNotif$, and so $incomingNotif(\langle\langle r,\vec{v}\rangle\rangle,(adaptNotif - t),i,\mathbf{N})$. Then by **Lemma 6.30** we have $incomingNotif(\langle r,\vec{v}\rangle,(adaptNotif - t - d),i,\mathbf{N})$ which is the same as $incomingNotif(\langle r,\vec{v}\rangle$,$adaptNotif$- $(t + d),i,\mathbf{N})$. □

Lemma 6.29 (instant_insuff_bad†)**.** *If a message with mode m was delivered exactly $adaptNotif$ time units ago by some entity, and the coverage of the message was insufficient with respect to said mode, and the entity in question has been broadcasting in mode m for some t time units greater than $adaptNotif$, then one of 3 things holds. Either a) A notification of the message and its coverage is pending immediately i.e. with timestamp 0 b) the entity in question is on the notification bad path or c) the entity is at the beginning of a transition to a fail-safe mode. This theorem captures an important behaviour of entities: reaction to coverage degradation with respect to the current mode. Whereas the reaction to insufficient coverage for next mode messages results in an abort of the mode transition, insufficiency of current-mode messages requires the more drastic measure of a transition to a fail-safe mode.*

Proof of instant_insuff_bad. Well, at the time of delivery of the message in question, we know from the semantics that a notification message was buffered with timestamp $adaptNotif$. Now, as the system evolves via delays, this timestamp decreases by the amount of the delay. Therefore by the time $adaptNotif$ time units have passed, as in this case, the timestamp reaches 0. So initially the first disjunct holds. From there, it can be shown that the only way this message can leave the notification queue is by being consumed by the software component, giving the base case for $notifBadPath$. It is possible then to show elsewhere that the entity either remains on the path or initiates a transition to a fail-safe mode. The transition to fail-safe mode is then preserved until time passes, which cannot be the case because one of the hypotheses is that the delivery happened exactly $adaptNotif$ time units ago. □

Lemma 6.30 (incomingNotif_del)**.** *If $\vec{v}$ is $incomingNotif$ with timestamp t, and the network can delay, then in the derivative network $\vec{v}$ is $incomingNotif$ with timestamp $t-d$. A useful corollary is that whenever $t=0$ delay is impossible. This follows directly from the semantics of the various languages involved.*

Proof of incomingNotif_del. This follows from the semantics of timed lists. The proof is similar to that of **Lemma 6.25**. □

Lemma 6.31 (incomingNotif_disc_pres†)**.** *If $\vec{v}$ is $incomingNotif$ with non-zero timestamp t, and the network performs a discrete action, then $\vec{v}$ is still $incomingNotif$ in the derivative network. This follows from the semantics of timed lists: discrete action cannot change the value of a timestamp, and only "timed out" messages may leave the list, i.e. those with a timestamp of 0. Together, these facts imply this result.*

Proof of incomingNotif_disc_pres. The proof is analogous to that of **Lemma 6.32**. □

Lemma 6.32 (outgoing_disc_pres†)**.** *If $\vec{v}$ is $outgoing$ for some entity with non-zero timestamp t, and the network performs a discrete action, then $\vec{v}$ is still $outgoing$ in the derivative network. This result is analogous in its statement and proof to **Lemma 6.31**.*

Proof of outgoing_disc_pres. This follows directly from the semantics of networks and timed lists. More specifically, only elements that have timed out are capable of leaving a timed list. The other elements can only leave the list after a delay equal to their timestamp has transpired, at which point their timestamp becomes 0. Hence, since the element in question has a non-zero timestamp, it cannot have left the list after only a discrete action, and so it is still in the output list of the derivative network. □

Chapter 7

The Coq Model of Comhordú

This chapter discusses the modelling of Comhordú in the proof assistant Coq. The specification language for Coq, Gallina, is a dependently typed functional programming language. All definitions and theorem statements are encoded in this language. Complementing this is the more imperative-style tactic language called Ltac, which is used to build proofs. In theory, every Coq development can be implemented using only Gallina; this is because of a result known as the Howard-Curry isomorphism, which in this context means that all proofs are seen as terms inhabiting a particular type- the type being isomorphic to the theorem statement. However, while Gallina is well suited to definitions, it often becomes unwieldy and unnatural when constructing proofs. Hence the language Ltac exists, which seems to resonate to a greater extent with human intuition when building a proof. In any case, the Coq engine eventually translates all developments in the Ltac language to Gallina specifications.

It would be possible to discuss the relative merits of the language of Coq in comparison to other languages, among other technical details of the Coq system. However, such in-depth discussions are outside the scope of this chapter. Instead, the aim here is to give a minimalistic description of the main features of Coq, just sufficient to allow the presentation of Comhordú in Coq. Of course, it is likely that some vital elements will be omitted from this short exposition; for example it may be that some terminology is used that was not properly defined. While this is ideally to be avoided, if it does occur the reader is referred to various external material which provides a good background on Coq.

The remainder of this chapter is broken down as follows. Section 7.1 introduces the notion of a Coq model, including various terms that will be used later in the chapter. Section 7.2 gives a bird's-eye view of the development of Comhordú in Coq. Section 7.3 is an account of some of the main issues/challenges encountered during the Coq modelling phase of this work. Section 7.4 gives a detailed exposition of features of the model which are of significant interest and/or importance. Section 7.5 outlines incomplete work and discusses strategies for its completion.

7.1 Introduction to Coq

Coq is a proof assistant in which theories or models can be encoded. A theory or model consists of a collection of definitions and results. It can be used for "pure" mathematical theories e.g. abstract algebra or applied models e.g. this coordination model. Encoding a model in Coq rather than on paper offers a number of advantages including rigour, ease of organisation and the potential for automation. On the other hand, the aforementioned rigour can introduce an often uncomfortable level of detail into specifications, rendering them quite verbose in comparison to their counterparts "on paper". A Coq model is essentially a series of object declarations written in a dependently typed functional programming language called Gallina. There are some minor variations, in that Coq has some special features for building proofs, but in the end the Coq model produced by the compiler is just a sequence of declarations.

What separates a Coq specification from a functional program written in a language like Haskell is the notion of dependent types, which adds a great deal of expressiveness to the language. The real distinguishing feature of Coq though is the distinction that is made between types and propositions. In Coq, there are two universes, the universe **Type** which contains all the possible types, and the universe **Prop** containing all the propositions, which are themselves a special sort of type. This separation is done in the spirit of a concept known as proof irrelevance, which basically means that any two proofs of the same proposition are in some sense equivalent. On the contrary, two elements of the same type are not necessarily equivalent e.g. the natural numbers 1 and 2 are distinct. A consequence of this is that anything defined in the **Type** universe cannot depend on the *structure* of a proof i.e. proofs are black box objects to anything outside of the **Prop** universe. The idea that proofs are themselves objects and the corresponding theorems types rests on a seminal result known as the Howard-Curry isomorphism, which draws a rigorous correspondence between the world of types/objects and the world of theorems/proofs. This topic could be discussed in more detail, but to do so would be superfluous for the purposes of this section.

The following is a non-exhaustive list, describing the main features and keywords of the Coq language. For a more in-depth introduction please see the literature.

Definitions There are a number of ways in which objects are defined in Coq. The two main ways are using the keywords **Definition** and **Inductive**. A definition declared with **Definition** in general defines a function, taking a number of arguments and building an object of the return type. Listing 7.1 is an example of a typical definition in Coq. In this case, the object being defined is a proposition i.e. an element of the **Prop** universe. The proposition is the statement that two sets, which are modelled in the Coq "ListSet" library as lists, are equal. Note that the object defined here is itself a type, not a proof of the equality of any two sets. The first arguments A is an arbitrary member of the **Type** universe, which is a good example of the expressive power of Coq to support polymorphism; functions need not be bound to rigidly defined types. It is enclosed in curly braces, meaning that it is implicit, and does not need to be specified in calls to the function; Coq has some in-built type inference that can deduce the type of A. The next two arguments are elements of the type "set A". At the top level, the returned proposition is a universal quantification over all elements of the type A, using the **forall** keyword. The body of this quantification then says that membership of the set l1 implies membership of l2 and vice versa.

Listing 7.1: A Typical Definition in Coq.

```
Definition set_equal {A : Type} (l1 l2 : set A) : Prop := forall a : A, a _:
    l1 <-> a _: l2.
```

A slightly more involved type of definition uses the **Inductive** keyword. This can be used to defined a type, a proposition, or a "family" of either. By a "family" of types (or propositions) what is meant is a collection of types each of which may be parametrised on one or more values. The key characteristic of the inductive definition is that the only elements of the types that exist are those that can be constructed by the rules given. In Listing 7.2, an inductive definition is shown which defines a family of types parametrised on a network, each called "reachableNet n" for some network n. The meaning of each such type is a set of traces showing that the network n is reachable. There are three rules for inhabiting these types. The "base case" rule, i.e. the rule that does not depend on a previous definition of "reachableNet", is "reachNetInit". This rule takes a proof of the initiality of a given network and returns an element of the reachable type for that network. Looking at this in terms of logical assertions rather than types, this can also be interpreted as saying that whenever a network is initial, then it is also reachable. However, this latter interpretation is a slight over-simplification in that it ignores the possibility of different elements of "initialNet n". The other two constructors are the "inductive cases" of this type. Each one takes a previously constructed representing the reachability of the network n, and a transition from n to some n', either discrete or timed, and returns an element representing the reachability of the network n'. These constructors can be viewed as nodes adding one step to the trace.

Listing 7.2: An Inductive Family of Types.

```
\label{lst:reachable}
Inductive reachableNet : Network -> Type :=
  | reachNetInit (n : Network) : initialNet n -> reachableNet n
  | reachNetDisc (n n' : Network) (a : ActDiscNet) : reachableNet n ->
    n -NA- a -NA> n' -> reachableNet n'
  | reachNetDel (n n' : Network) (d : Delay) : reachableNet n ->
    n -ND- d -ND> n' -> reachableNet n'.
```

Results In Coq the keywords **Theorem**, **Lemma**, or **Corollary** are used to specify results. There is no distinction between these different keywords except the meaning inferred by the programmer, where theorems are usually more important than lemmas and corollaries are results that follow on from other results. Each result consists of two parts: statement and proof. The statement is some proposition, while the proof is a term inhabiting this proposition. For a given proposition, it is conceivable that there may be many proofs. Usually, the actual proof itself is irrelevant in a certain sense, in that all that is of importance is that there is *some* element demonstrating the truth of a theorem. In fact, this point of view has a direct realisation in Coq with the library "Coq.Logic.ProofIrrelevance", which gives as an axiom that any two proofs of a given proposition are equal. Of course, structurally they may not be so, but the point is that at a certain level of abstraction, the structure of proofs doesn't matter. It is for this reason that the **Prop** universe and the **Type** universe are made distinct by Coq: otherwise proof irrelevance would have to apply to all objects, leading to the absurdity that any pair of elements are equal.

Now, proofs are, technically speaking, objects inhabiting their corresponding types, and can be defined using the **Definition** keyword in the standard way. However, they are most often defined using what is known in Coq as "proof editing mode". This mode resonates with the usual style of proofs as trees, or more correctly acyclic directed graphs, starting with some hypotheses and using rules of inference to build towards some results. It is hard to imagine how some proofs could be done without this mode i.e. using purely declarative means. Even if they were to be done this way, the resulting terms would no doubt in most cases be unwieldy and verbose. In proof editing mode, the user is presented with the goal, which is some type, and a list of hypotheses, which are pre-existing local or global objects of various other types that are assumed to exist, either in the context of this theorem or just in general. Using rules of inference, previously proved results, and tactics, the user then has two choices: to work forwards from the hypothesis building towards the goal or to reduce the goal to previously proved results and/or hypotheses. The system is geared more towards the latter, though experience with this model has seen that a mixture of both is usually present in a given proof. As the goal is reduced, or various tactics are applied, the goal may be split into sub-goals. Eventually, when all the sub-goals have been proved, the proof is done, and the user ends with the command **Qed**, at which point the proof is saved and converted by the underlying engine of the theorem prover to a function mapping the hypotheses to the goal.

An example of a simple lemma is given in Listing 7.3. The **Lemma** keyword precedes the name of the lemma, in this case "Rplus_le_swap_rr", which is followed by a colon and then the lemma statement or equivalently its type. The statement first quantifies over 3 real numbers x, y and z, using the **forall** keyword. It then posits that whenever x is less than or equal to the sum of the other two terms, then x minus z is less than y. This fact is obvious from the known rules of algebra; still, this is a good example of a result in Coq that is simple enough to explain the main mechanisms of proof mode. After the theorem statement comes a series of tactics, ending with the keyword **Qed**. After the application of each tactic, the proof context gets transformed until eventually the only remaining goal is solvable by the tactic "ring", which solves equations in real numbers. Limitations of space here preclude a full exposition of each step of the theorem, including all the hypotheses and the sub-goals.

A brief explanation of the steps of the theorem here follows. First, the intros tactic assumes the variables

x, y and z into the context of the hypotheses, along with the premise that x is less than or equal the sum shown, leaving the final inequality as the goal. The "eapply" tactic allows an unknown term to be added to each side of the equation- this term can be filled in at a later stage. Next, transitivity is applied: this fixes the aforementioned previously unknown term to be z. The result "Req_le" is then used to convert the remaining inequality into an equation, which is then solved by the automatic "ring" tactic, ending the proof. These steps may be hard to follow without the intermediate proof contexts. The interested reader is encouraged to download Coq, along with the IDE and use it to step through this proof in more detail.

Listing 7.3: A Simple Lemma involving Inequalities.

```
Lemma Rplus_le_swap_rr : forall (x y z : R),
  x <= y + z -> x - z <= y. intros. eapply Rplus_le_reg_r.
  eapply Rle_trans;[ | apply H ].
  apply Req_le. ring. Qed.
```

Assumptions The previous result about inequalities shows just how much detail is involved in even the simple Coq proofs. One way around this is to use more and more automation. However, this is not always possible, or it is not immediately obvious how to do so. Also, sometimes it is desirable to be able to state a result now and use it, leaving the proof until later. For this reason, Coq allows assumptions to be made with the keywords **Parameter**, **Conjecture** and **Axiom**. There are other keywords, but these are the main ones used in this model. An assumption is a name given to an element of a type that is assumed to exist without its explicit construction. All assumptions share this essential property; however the convention is adopted here that they are prefixed with different keywords to convey meaning to the reader as follows. The **Parameter** keyword is usually reserved for something non-proof object that needs to be defined later on. Anything defined with the word **Axiom** is intended to be left as it is, an uninterpreted, assumed element of some type whose structure is intentionally left abstract and is not meant to be refined, at least not in the current version of the model. A definition using **Conjecture** is usually some result that is deemed obvious.

Another keyword that is used throughout this model is **Admitted**. This appears anywhere after the statement of a result in proof mode, essentially "short circuiting" the proof and instead assuming the result in an abstract way. There is technically no difference between this and using one of the previously mentioned keywords. Nevertheless, on a conceptual level, this keyword is used for things that are intended to be defined fully at some stage, particularly if it interrupts a partially complete proof, as is the case for a handful of results. Overall, this ability to assume certain elements of the model without their explicit construction is an essential feature without which the development of the model would have been intractable.

Ltac Tactics in Coq are means of progressing a proof in proof mode. The Coq distribution is accompanied by a standard distribution of in-built tactics. These can be combined into more complex user-defined tactics, which are defined with the keyword **Ltac**. The goal of developing such tactics is to increase the level of automation in proofs i.e. they are meant to replace user input wherever possible. This confers the advantages of both increased succinctness in the proof scripts and reduced time and effort for the human being who has to perform the proof. The downside is that, with increased automation, scripts can be slower to process by the machine: consider, for example, an automation tactic that tries a number of different options in a brute force manner vs. a user who uses intuition to guide them along a more tailored approach. Also increased automation means the underlying structure of proofs can become more uncertain- remember every proof built with tactics is eventually converted to a proof object by the Coq engine. Nonetheless, these disadvantages are minor in comparison to the gain brought by automation, and are of little concern here. For examples of tactics, see Section 7.4.2 and Section 7.4.6.

Notation The expression of certain terms in Coq involving functions can often be quite an ordeal and can

seem quite unnatural. To alleviate this somewhat, notations can be defined. Consider the example of Listing 7.4. Here, set equality between the sets a and b is given the notation "a ={}= b". The keyword **Notation** declares the notation, which follows in quotation marks. To the right of the notation is its meaning, which is the function "set_equal" that was defined earlier, applied to the arguments a and b. Notations are certainly not a necessity but they are convenient, and they do feature to a reasonable extent in this development.

Listing 7.4: A Notation for Set Equality.

```
Notation "a ={}= b" := (set_equal a b) (at level 70).
```

match A key feature of Coq is pattern matching. This is used in the definition of functions and tactics. A pattern matching clause begins with the keyword **match** and ends with the keyword **end**. The object being matched is compared against a number of patterns, and different actions/results can be taken/given depending on the "shape" of the matched patterns. The reader is referred forward to Listing 7.9 for an example of pattern matching.

let clause A let clause allows certain cumbersome terms to be removed from an expression and replaced with more succinct representations. For example, in Listing 7.5, an interval is defined in terms of a lower and upper bound. Both bounds are reasonably lengthy expressions themselves, so they are given the pseudonyms "upper" and "lower", making the final term in the definition more succinct. Like many of the constructions discussed thus far, these let clauses are not strictly necessary; rather they are used to enhance the tidiness of code.

Listing 7.5: Use of **let** Clauses.

```
(** The interval of time in the past during which we can show a
sufficient pre-broadcast takes place, parametrised on the mode m.*)
Definition preDeliveredInterval (m : Mode) : Interval :=
  let lower := Rmax adaptNotif transMax in
  let upper := lower + period m + trans m in
  (\lower, upper\].
```

7.2 Overall Structure

Here the collection of files comprising the Coq model are described. At the time of writing, the Coq model of Comhordú can be found online at: https://www.scss.tcd.ie/ bhandalc/CoqDoc/toc.html. There are 23 files in total comprising the Coq model. These can be sorted into categories as per the following description. The division of the model across multiple files is essential, in part due to its sheer bulk of almost 10,000 lines of code, but also for conceptual clarity, modularised compiling capability and ease of debugging; there is further discussion on these issues in Section 7.3. Each item in the following description covers one or more of the constituent files, outlining its main features and its contribution to the overall model. Specific details from the files are not discussed here, but a select few of particular interest appear in Section 7.4.

General Tactics The file "GenTacs.v" contains generic tactics for making the proofs more concise. Some of these are truly generic e.g. eliminating conjunctions; others are more specialised for this development, nevertheless they do not rely upon any Comhordú specific elements for their definition. These tactics do not add any insights to the model or its proof, they simply aid in speeding up the proof process. Some tactics found here form a reusable set that any Coq user might use.

Standard Results In the file "StandardResults.v" a number of results are proved or stated that are not specific to the Comhordú model. The file chiefly consists of results about real numbers and their inequalities,

positions, distances and intervals. The results are simple, and many would normally be assumed or deemed obvious.

Comhordú Basics This file "ComhBasics.v" contains various constants relating to the Comhordú model, along with certain relationships between these constants. Also some basic definitions are to be found here such as the notion of a network of entities. This file can be seen as the foundation for the definition of the Comhordú model and protocol.

Languages There are a number of files dedicated to the definitions of the syntax and semantics of the languages that are used to describe Comhordú systems. Recall these languages from Chapter 4. "LanguageFoundations.v" lays the foundation for the software language file by defining an expression language and various other low level components. "SoftwareLanguage.v" defines a CCS style language with point to point value passing communication and time; although it ideally would be defined elsewhere the protocol definition is also included here for reasons discussed later. "InterfaceLanguage.v" gives a language that models the environment and the underlying Space-Elastic Model. "ModeStateLanguage.v" captures the behaviour of the mode state component of an entity. "EntityLanguage.v" composes the software, interface and mode state languages to form one higher level entity language. "NetworkLangauge.v" defines a language for groups of entities, called networks, in particular introducing laws for communication between entities and movement of entities in space.

Protocol Auxiliaries The file "ProtAuxDefs.v" defines a number of state predicates over software processes: these are essentially symbolic abstractions used to circumvent the infinitude of the LTS for the protocol process. The file "ProtAuxResults.v" contains a number of results, mostly relating to the aforementioned state predicates, such as preservation of state predicates by the delay transition.

Entity Auxiliaries Most of the file "EntAuxDefs.v" is dedicated to lifting state predicates from the software level to the entity level. A more interesting contribution of this file is the definition of the path predicates, which are used in later proofs to reason about reactions to events. The file "EntAuxResults.v" can be subdivided into a number of sections: results about the relationships between the various components of an entity, delay preservation of state predicates, urgency of states i.e. states in which no time can pass, and results that are lifted from the protocol auxiliary level.

Basic Network Auxiliaries Certain simple auxiliary definitions about networks are defined here for the ease of specification and proof of properties at a higher level. Also, a number of definitions/results pertaining to state predicates are lifted from the entity level to this level.

Network Auxiliaries The file "NetAuxDefs.v" concerns itself primarily with the definition of the various history relations that are key components of the overall proof. Results about these relations are then split into a number of files; usually each of these is loosely themed around some important result and contains some smaller results. These files are: "NARIncomp.v", "NARInsuff.v", "NARMisc.v", "NARMsgPosition.v", "NARNonFS_currSince.v", "NAROvlpTime.v" and "NARTop.v". The file "NARTop.v" is the top level file of this group, tying together results from the other files.

Main This file contains the top level safety proof. It imports results and definitions from the aforementioned files in order to construct this proof. If further results, beyond safety, were to be proved about Comhordú, then they would belong here. However, for now safety is the only high level result that exists here.

7.3 Coding Challenges

The task of coding the Comhordú model, protocol and proof in Coq poses a number of challenges that need to be resolved. Any implicit assumptions and shortcuts inherent in a model sketched by hand cannot exist in a precise programming language. This has its merits: there can be absolutely no ambiguity in code, which unlike natural

language has a unique meaning. However, the price paid is verbosity: the rigour of a well defined language forces the programmer to spell out in full detail every assumption and define every component exactly, even elements that would normally be deemed "obvious" or left out completely as implicit assumptions. This section discusses some of the issues that have been encountered in the coding of this model, along with the solutions that are adopted to address these issues. Much of the discussion to follow focuses on the (im)practicality of coding in Coq.

File Modularisation It was obvious from the scale of the Comhordú model on paper that the Coq model would need to be modularised i.e. split across a number of files. Still, the exact nature of this modularisation did not follow directly from the model that had been sketched out. This led to a chicken-and-egg scenario: it was desirable to have a skeleton of the file layout before beginning to code; on the other hand, this structure would only become apparent during the coding process itself. The only solution to such a problem is to make the best informed estimate of the layout, and then begin coding, allowing for later changes via flexibility in the overall structure. Hence the code was arbitrarily split in a reasonably logical manner before coding began, and split again a number of times as the size of certain files grew large. The result is the current file layout as presented in Section 7.2. There are conceivably "better" layouts than this one, but it suffices for the task at hand.

Verbosity of Coq Because it is precise and low level, Coq is quite verbose. This is true of definitions, theorem statements and proofs themselves. There are a number of techniques to circumvent this verbosity, used to various degrees in the Comhordú model in Coq. One such technique is to abstract, employing the generality offered by abstraction as a means of avoiding re-iterating many similar definitions/proofs.

Another antidote to verbosity is the use of tactics, which help reduce the size of proofs and make them more robust. They are particularly powerful tools in that they offer abstraction and reusability of proof steps. A number of tactics exist in the Comhordú model, to varying degrees of generality. It is arguable that more tactics could have been developed to further reduce the verbosity; nonetheless there is an overhead to writing tactics as opposed to simply proceeding with a proof and there is no set rule to when this overhead will pay off: this intuition develops with experience.

Stubs are proof statements whose proof has been omitted, i.e. propositions. These can be considered as a shortcut around the verbosity of proofs. However, the machine-verified guarantee is then lost.

Constructivism The logic underlying Coq is constructivist. This means that in order to prove the existence of some object an explicit construction of that object must be given. It is not enough to assume the object doesn't exist and derive a contradiction from this i.e. a reductio ad absurdum. This restriction is characterised by the omission of a certain standard law from classical logic, the law of the excluded middle, which asserts that a proposition P or its negation $\neg P$ must be true. To omit such a law seems counter intuitive: surely something is either true or it is not true? However, the reasons for leaving it out are rooted in the constructivist nature of how proofs are done in Coq: they can be seen as terms inhabiting types corresponding to the theorem statement. They can also be seen as functions mapping previous results to the current one. Proofs by contradiction don't give such an explicit mapping beyond asserting that it exists and so are not allowed in constructivist systems like Coq. One workaround however is to assume stubs for particular instances of the law of the excluded middle. This is the approach that has been adopted. Another option would be to simply assume as an axiom the law of the excluded middle. However, this approach seems like it would encourage a lazy attitude to proofs and so only instances of the law are assumed and at that only when absolutely necessary.

When to Automate Automation is a key advantage of using a proof assistant such as Coq. It involves the use and development of what are called tactics. These are imperative routines that are intended to replace human input whenever possible. When used cleverly, tactics can reduce both the time taken to develop Coq objects and the verbosity of their specification. Furthermore, they tend increase the robustness of

proofs i.e. their resilience to small changes. They are chiefly used in proofs, but technically can be used in the definition of non-proof objects also, due to the isomorphic nature of proof and object in Coq. While there are clearly advantages to using tactics in Coq, there are also some drawbacks. Tactics development imposes an overhead on the coding process, both in terms of space and time. Hence tactics are only practical if this overhead is compensated considerably in terms of reduced coding time and more succinct proofs/definitions. This is an example of the classic "Productivity vs. production" problem, and the decision to use or to not use tactics in a particular scenario is ultimately only guesswork informed by experience. In the Coq model of Comhordú, older proofs contain considerably less automation and are clearly more verbose and brittle. More recent proofs contain a greater number of tactics and are constructed in a more robust way.

When to Abstract In a similar vein to automation abstraction offers the advantage of reusability and hence reduced coding time. It also provides conceptual elegance and simplicity. However, again like in the case of automation, there is no formulaic way of determining when abstraction is necessary, let alone what the abstraction should be. It is felt that the current model could be more abstract in a number of ways e.g. to name a few: time could be modelled axiomatically, history relations could be defined in terms of a common structure, an abstract characterisation could be used for labelled transition systems and instantiated for the specific languages involved. The reason such abstractions are not present in the current model is that they have only become apparent after modelling has been done. This illuminates another feature of abstraction: it often only becomes apparent after a number of specific instances have been done.

7.4 Content Highlights

In this section, a select number of features from the Coq model are listed and discussed. The entire model, consisting of close to 10,000 lines of code, would be impossible to discuss here, given that the discussion of code often matches or even exceeds the size of the code itself. Hence the exposition is limited to only important or interesting elements, or "typical" elements that epitomise certain classes of structures or coding techniques. These "elements" comprising the Coq code are either tactics, theorems, definitions or some form of assumptions e.g. axioms. The idea here is to provide enough code to give an idea of the flavour of the overall model. Section 7.4.1 lists some of the axioms underlying the model; Section 7.4.2 contains some general tactics; Section 7.4.3 briefly presents some elements of the software fragment of the overall language; Section 7.4.4 gives some of the protocol definitions; Section 7.4.5 discusses and presents a selection of state predicates; Section 7.4.6 details specialised tactics that were written specifically for this model. In the code listings to come, often snippets of code are broken by dots "…", indicating that there is some code omitted in the listing.

7.4.1 Some Underlying Axioms

A number of axioms are assumed to underlie the Comhordú model. These are declared in Coq using the "Axiom" keyword, which allows for the assumption of an element of some type without explicit construction of that element. Of course, this is contrary to the principle of constructivism underlying Coq, and so every axiom that is assumed increases the vulnerability of the model to inconsistency. It is thus desirable to minimise the number of such assumptions. Nevertheless, the following assumptions are necessarily axiomatic in nature: to define them would be to restrict the generality of the model, which is not desirable. The entire set of axioms is not given here, just a representative subset. Accompanying each axiom is the Coq comment describing it.

Let us now explain some these axioms. The first defines a constant $adaptNotif$ which has the type "Time". The axiom "fsDecMode" is defined in terms of two previously existing functions "decidable" and "failSafe". To say a proposition is decidable is to say that there is always a proof of either the proposition itself or its negation.

The predicate "failSafe" asserts that a mode is a fail-safe mode. The axiom "failSafeSucc" constitutes the assumption of a function from modes to modes. The intuitive meaning is that every mode is assumed to have a fail-safe successor, i.e. a fail-safe mode to which it can transition. The next axiom states that the fail-safe successor to a mode is indeed fail-safe. The final axiom says that the function "minDistComp", assumed in the previous axiom, is symmetric; that is, the order of its arguments can be switched without changing its value.

Listing 7.6: Some sample axioms.

```
(** The time it takes for an entity to be notified of a message delivery area.*)
Axiom adaptNotif : Time.
...
(** A mode is either fail-safe or it isn't.*)
Axiom fsDecMode : forall (m : Mode), decidable (failSafe m).
...
(** We assume that every mode transitions to some fail-safe mode,
and this function simply arbitrarily picks one such fail-safe mode.*)
Axiom failSafeSucc : Mode -> Mode.
...
(** This is the assumption that failSafeSucc m is indeed fail-safe for all m.*)
Axiom failSafeSuccFS : forall (m : Mode), failSafe (failSafeSucc m).
...
(** The minimum distance of compatibility function is assumed to exist.
It tells us the minimum distance by which two modes must be separated in
order for them to be compatible. If two entities in these modes are
separated by a distance greater than or equal to this distance,
then they are compatible.*)
Axiom minDistComp : Mode -> Mode -> Distance.
...
(** The minimum distance of compatibility function is symmetric.*)
Axiom minDistCompSymmetric : forall (m1 m2 : Mode),
  minDistComp m1 m2 = minDistComp m2 m1.
```

7.4.2 General Tactics

Some general tactics have been developed to aid with proofs; general in the sense that they are applicable to Coq code outside of the Comhordú model. Indeed, many of these tactics are reusable for the general Coq coder. The tactics are for the most part simple, but they aid considerably in reducing the bulk of code in the theorems because they are used quite often. These tactics are not guaranteed to be unique- some may exist in some repository that another person has written. However, many are so simple that it was more convenient to just write them rather than search for them.

The "swap" tactic is listed in Listing 7.7. Since this is the first tactic to be defined here, let us first discuss some of the notation. The keyword "Ltac" is used to declare a tactic. Following this is the name of the tactic and any number of arguments. After this is the definition symbol ":=" and then the tactic body.

In this case there are two arguments A and B. The meaning of the tactic is to swap the names A and B i.e. rename one object into the other. This is done by first renaming A into a fresh variable H, which has been created by a "let" clause as shown; then B is renamed into A, and finally H into B. The use of a fresh name is a common feature of tactics, making them more robust than tactics that use set names that may clash with existing names in the list of hypotheses. Observe the semi-colons ";" separating various parts of the tactic definition. This denotes sequential composition: i.e. the terms separated by the semi-colons are tactic commands that are to be executed in sequence. The semi-colon operator is actually slightly more complicated than this, but in this case it suffices to think of it as sequential composition.

Listing 7.7: The swap tactic.

```
Ltac swap A B := let H := fresh in rename A into H; rename B into A; rename H
    into B.
```

The tactic shown in Listing 7.9 is inspired by a tactic called "applys_eq" from the library "LibTactics.v". It tries to apply a hypothesis H with 3 arguments to a goal with 3 arguments, leaving only the non-trivial equalities of the matched arguments as sub-goals. There are similar tactics for different numbers of arguments. Clearly, a generalised tactic taking the number of arguments as a parameter is desirable here; indeed such a tactic is provided in "LibTactics.v". However, for unknown reasons the tactic provided there did not work with this model, and the collection of tactics with "fixed" numbers of arguments sufficed as the most practical solution.

In order to leave only the non-trivial equalities as spawned sub-goals, the tactic "replace_nontriv" is employed. This is defined in Listing 7.8. The tactic body begins with an application of the "replace" tactic, which generates two sub-goals. The first sub-goal is the current goal with the specified substitution applied, the second is the equality between the substituted term and the new term. The semi-colon operator is used here in a different way to before because the tactic preceding it spawns two goals: the composition modelled by this operator is in general branching rather than just sequential, which is a special case. Different tactics may be used on different sub-goals; these are enclosed in square brackets and separated by vertical bars. An empty placeholder for a tactic signifies the identity tactic or "idtac" which essentially does nothing to the goal, as is the case for the first tactic here. The second tactic tries to solve the goal trivially with the in-built "reflexivity" tactic which performs some simplification and then tries to match the goal to an equality of the form "a = a". The "try" keyword here indicates that failure of the reflexivity tactic does not cause the entire top level tactic to fail, but rather results in the identity tactic being applied i.e. the equality remains to be proved manually.

Listing 7.8: A specialised replace tactic.

```
Ltac replace_nontriv x y := replace x with y; [ |try reflexivity].
```

Listing 7.9: Apply and Replace.

```
Ltac my_applys_eq3 H :=
  match goal with
  | [ |- ?F ?x1 ?x2 ?x3] =>
    match type of H with
    | (?G ?y1 ?y2 ?y3) =>
      replace_nontriv x1 y1; replace_nontriv x2 y2;
      replace_nontriv x3 y3
    end;
    [apply H |..]
  end.
```

7.4.3 Syntax and Semantics of the Software Fragment

The software fragment of the overall language has been selected for presentation here; the other fragments of the language are similar. Note that the aim here is not to give the full presentation of the language in terms of its syntax and operational rules. For this, the reader is referred either to the full Coq model or to the presentation of the language in Chapter 4. Rather, the aim here is to portray the typical features of such a language encoded in Coq.

Listing 7.10 depicts the syntax of the language. The "Inductive" keyword specifies that all terms of the language must be built with a finite application of the constructors that follow. These constructors match the terms of the language syntax as put forth in Chapter 4. A vertical bar separates each constructor from the next. Constructors are usually placed on separate lines from each other. Arrows of the form "$->$" separate unnamed arguments to a particular constructor, with the final term in such a sequence constituting the constructed object.

For example, the "outPrefix" constructor takes three arguments: a channel, a list of expressions and a process term and constructs a process term.

Some of the semantics of this language are presented in Listing 7.11 and Listing 7.12. Again, this is not an exhaustive explanation of the workings of the semantics but rather an exposition of how they are typically encoded in Coq. The term "p1 + p2" in the discrete semantics is simply shorthand for the sum process. Notice that the conclusion to each rule is an element of a type in the inductive family of types being defined. For example, "stepDiscProc" defines a family of types with three parameters, and the constructed type at the end of the rule "stepDpThen" is "stepDiscProc (ifThen b p) d p"', which is one of the types in this family.

Listing 7.10: Software Syntax.

```
Inductive ProcTerm : Type :=
  | nilProc : ProcTerm
  | outPrefix : Channel -> list Exp -> ProcTerm -> ProcTerm
  | inPrefix : Channel -> list Var -> ProcTerm -> ProcTerm
  | ifThen : BoolExp -> ProcTerm -> ProcTerm
  | sum : ProcTerm -> ProcTerm -> ProcTerm
  | parComp : ProcTerm -> ProcTerm -> ProcTerm
  | app : Name -> list Exp -> ProcTerm
  | del : Exp -> ProcTerm -> ProcTerm.
```

Listing 7.11: Discrete Software Semantics: Sample.

```
Inductive stepDiscProc : ProcTerm -> DiscAct -> ProcTerm -> Prop :=
  ...
  | stepDpThen : forall (b : BoolExp) (p p' : ProcTerm) (d : DiscAct),
    stepDiscProc p d p' -> evalBoolExpFunTot b = true ->
    stepDiscProc (ifThen b p) d p'
  ...
  | stepDpChoiceR : forall (p1 p2 p' : ProcTerm) (d : DiscAct),
    stepDiscProc p2 d p' -> stepDiscProc (p1 $+$ p2) d p'
  ...
  | stepDpSyncLR : forall (p p' q q' : ProcTerm) (c : Channel) (l : list Base),
    stepDiscProc p (outAct c l) p' -> stepDiscProc q (inAct c l) q' ->
  ...
  | stepDpTimeOut : forall (p p' : ProcTerm) (d : DiscAct) (e : Exp),
    stepDiscProc p d p' -> evalExp e zeroTime -> stepDiscProc (del e p) d p'.
```

Listing 7.12: Timed Software Semantics: Sample.

```
Inductive stepTimedProc : ProcTerm -> Delay -> ProcTerm -> Prop :=
  | stepTpNil : forall d : Delay, stepTimedProc nilProc d nilProc
  ...
  | stepTpSum : forall (p1 p2 p1' p2' : ProcTerm) (d : Delay),
    stepTimedProc p1 d p1' -> stepTimedProc p2 d p2' ->
    stepTimedProc (sum p1 p2) d (sum p1' p2')
  ...
  | stepTpDelSub : forall (p : ProcTerm) (d : Delay) (e : Exp) (t : Time)
    (H : d <= t), evalExp e t ->
    stepTimedProc (del e p) d (del (minusTime t d H) p)
```

7.4.4 Protocol Component Definitions

Given here are a few definitions from the protocol specification in Coq. Hopefully these will give enough understanding to grasp the entire specification. Listing 7.13 defines a countably infinite list of variables indexed

by natural numbers. Notice the "var" constructor used to "wrap" a natural number and produce a variable. Since referring to variables using this var constructor can be cumbersome and difficult to read, a number of syntactic shortcuts for variables are used. Some of these are shown in Listing 7.14. Listing 7.15 provides more notational definitions. These are succinct ways of expressing the software language syntax by use of symbols instead of constructor applications. These notation definitions begin with the keyword "Notation", followed by a term in quotation marks containing some meta-variables and symbols. Then there is a definition symbol ":=" and to the right of this is the meaning of the notation just declared, which in this case is always a constructor of the language. The notation definitions end with some information enclosed in parentheses about the binding level and associativity, which is necessary for the Coq parser but is of little interest here.

Listing 7.13: Variables.

```
Inductive Var : Type:=
  | var : nat -> Var.
```

Listing 7.14: Variable Notations.

```
Definition vM := (var 0).
...
Definition vL := (var 4).
...
Definition vX := (var 9).
```

Listing 7.15: Software Syntactic Sugar.

```
Notation "c $< l >$ ? P" := (inPrefix c l P) (at level 41, right associativity).
Notation "c $< l >$ ! P" := (outPrefix c l P) (at level 41, right associativity).
Notation "c ? P" := (inPrefix c [] P) (at level 41, right associativity).
Notation "c ! P" := (outPrefix c [] P) (at level 41, right associativity).
Notation "B >> P" := (ifThen B P) (at level 41, right associativity).
Notation "P $+$ Q" := (sum P Q) (at level 50).
Notation "P $||$ Q" := (parComp P Q) (at level 45, right associativity).
Notation "h $( l )$" := (app h l) (at level 41, right associativity).
Notation " $< E >$ P " := (del E P) (at level 41, right associativity).
```

The definition of the first component of the protocol, namely the broadcast component is shown in Listing 7.16. The first term "bcWait" takes two expressions as parameters. The first represents the mode to be broadcast, while the second represents the remaining time. This process is waiting to broadcast a message. The other definitions are "sleeping" and "bcReady". Notice the use of the choice operator "+" and the delay prefix "$< x >$". The parameter list to a process is given via square bracket (list) notation as in "bcReadyN ($[m, eVarvL]$)". In the definition of "sleeping", a variable "vM" is bound to an input on the channel "chanWake". This variable is then used as an argument to the process "bcWaitN". The function "eVar" simply wraps a variable term to form an expression term. The meaning of these terms is not explained here, but is covered in Chapter 5. Snippets of code from the overlap and listener component definitions are given in Listing 7.17 and Listing 7.18 respectively.

Listing 7.16: Broadcast Definitions.

```
Definition bcWait (m x : Exp) := chanTrans? sleepingN $+$ $<x>$chanPos$<vL>$?
    bcReadyN $([m, eVar vL])$ $+$ sleepingN.

Definition sleeping := chanWake$<vM>$? bcWaitN $([eVar vM, liftTimeExp zeroTime])
    $ $+$ chanTrans? sleepingN.
```

```
Definition bcReady (m l : Exp) := chanOutProc$<[m, l]>$!bcWaitN $([m, (ePeriod m)
    ])$.
```

Listing 7.17: Select Overlap Definitions.

```
Definition dormant := chanMNext$<vM'>$? initN $(eVar vM')$ $+$ chanAbort?
    dormantN $+$ chanBad? tfsStartN.
...
Definition ovWait (m' t x y : Exp) := chanAbort? ovAbortN $+$ chanBad? tfsStartN
    $+$ $<y>$chanPos$<vL>$? ovReadyN $([m', t, eVar vL])$ $+$ $<x>$chanPause!
    switchBcN $(m')$.
...
Definition ovReady (m' t l : Exp) := chanOutProc$<[m', l]>$!ovWaitN $([m', (
    eSubtract t (ePeriod m')), (ePeriod m')])$.
...
Definition tfsNext (m : Exp) := chanMNext$<eFailSafeSucc m>$!tfsCurrN $+$
    chanMStable!tfsNextN $(m)$.
...
Definition ovAbort := chanMStable!dormantN.
```

Listing 7.18: Select Listener Definitions.

```
Definition listening := chanAN$<[vR, vM'']>$? checkRangeN $([eVar vM'', eVar vR])
    $ $+$ chanInProc$<[vM'', vL']>$? gotMsgN $([eVar vM'', eVar vL'])$ $+$
    chanPause? pausedN $+$ chanUnpause? listeningN.

Definition checkRange (m'' r : Exp) := (bSufficient (m'') (r)) >> listeningN $+$
    !~(bSufficient (m'') (r)) >> rangeBadN $(m'')$.
...
Definition badOvlp := chanBad!listeningN.

Definition abortOvlp := chanAbort!listeningN.
...
Definition paused := chanUnpause? listeningN.
```

Remark 7.1 (Modelling Recursion)**.** To add recursion to the language, a number of names are introduced, along with a definition relation mapping these names to process bodies. These allow for finite, inductive terms to exhibit infinite behaviour because process bodies can contain names and names in turn can map to bodies. Without having these two distinct layers, recursion would still be possible, via either coinductive process bodies or a recursion operator. There is no "correct" choice, though it is believed that the current one is more intuitive than a recursion operator and it allows for proofs by induction, a feature that would be lost with coinduction. ▲

7.4.5 State Predicates

A large portion of the Coq model is taken up by what are referred to here as "state predicates". These are predicates on software terms, possibly with parameters of other types, that assert that a process term is of a certain form. There are two broad categories of state predicates. "Simple" state predicates are used to capture each definition of the protocol processes. "Composite" or compound state predicates are at a higher level, asserting that a process is in one of a number of states. At this point, the question may arise as to why such state predicates are necessary at all? The answer to this lies in the fact that the LTS of the protocol is uncountably infinite in its number of states, due to the parametrisation of processes with parameters drawn from infinite types e.g. the type Time. The state predicates chop this infinite state-space into a finite number of groups,

each containing of course an infinite number of states. Each state predicate corresponds to one such group, containing all processes of a certain "shape", so to speak, differing only in their parameter values. In a similar vein to symbolic model checking, then, these state predicates allow the infinite state-space to be reasoned about in a finite way, making it amenable to reference by the various theorems making up the proof of safety.

Depicted in Listing 7.19 is the state predicate for the state "bcWait". Like most simple state predicates, there are two constructors. The first constructor says that the application of the name "bcWaitN" to some pair of expressions constitutes a bcWaitState. The second constructor says that any process which is "bcWaitBody" applied to a pair of expressions is a bcWaitState. These two cases look very similar, and raise the question, why is it necessary to have two distinct cases at all? The answer to this is partly rooted in recursion, which is discussed in Remark 7.1. However, this does not give the full picture. For a small number of the state predicates, there is a subtle difference between the process body definition and the process name application term. For example, in Listing 7.20, notice that the first constructor takes only 3 expressions, while the second takes 4. This is because the time variable passed to the process body is in fact duplicated, in one instance as a parameter to a nested application term and in another instance as a delay guard. When time passes, the delay guard begins to expire but the parameter in the nested process does not. Hence they become distinct. For this reason, an extra variable is needed to capture this more general shape.

Listing 7.19: Typical State Predicate.

```
Inductive bcWaitState (m : Mode) (x : Time) : ProcTerm -> Prop :=
  | bcwaitName (e1 e2 : Exp) : e1 |_| m -> e2 |_| x ->
    bcWaitState m x (bcWaitN $([e1, e2])$)
  | bcwaitBody (e1 e2 : Exp) : e1 |_| m -> e2 |_| x ->
    bcWaitState m x (bcWait e1 e2).
```

Listing 7.20: Overlap Wait State Predicate.

```
Inductive ovWaitState (m : Mode) (t : Time) : Time -> Time -> ProcTerm -> Prop :=
  | ovwaitName (e1 e2 e3 : Exp) (y : Time) : e1 |_| m -> e2 |_| t ->
    e3 |_| y -> ovWaitState m t t y (ovWaitN $([e1, e2, e3])$)
  | ovwaitBody (e1 e2 e3 e4 : Exp) (x y : Time) : e1 |_| m -> e2 |_| t ->
    e3 |_| x -> e4 |_| y -> ovWaitState m t x y (ovWait e1 e2 e3 e4).
```

While most of the state predicates are relatively straightforward, following the basic pattern previously outlined, some are a little more involved. An example of such a predicate is given in Listing 7.21, in which a "listeningState" is characterised. In this case, the usual constructors are included, i.e. name and body, but there are also other constructors. These additional constructors correspond to process terms that "evaluate" to listeningState via conditional statements. What this means is that listeningState is either in the "then" or "else" branch of the conditional characterised by the process in question, and the condition is either "true" or "false" respectively; hence the process essentially behaves as the listeningState in these cases. For example, "listenCrBody" takes as a parameter that "sufficient m r" holds, and yields an assertion of the listeningState predicate for the conditional term "checkRange e1 e2", whose condition evaluates to "sufficient m r".

Listing 7.21: Listening State Predicate.

```
Inductive listeningState : ProcTerm -> Prop :=
  | listenName : listeningState listeningN
  | listenBody : listeningState listening
  | listenCrName (m : Mode) (r : Distance) (e1 e2 : Exp) :
    sufficient m r -> e1 |_| m -> e2 |_| r ->
    listeningState (checkRangeN $([e1, e2])$)
  | listenCrBody (m : Mode) (r : Distance) (e1 e2 : Exp) :
    sufficient m r -> e1 |_| m -> e2 |_| r ->
    listeningState (checkRange e1 e2)
```

```
| listenNeName (m m' : Mode) (e1 e2 : Exp) :
  m <> m' -> e1 |_| m -> e2 |_| m' ->
  listeningState (nextEqN $([e1, e2])$)
| listenNeBody (m m' : Mode) (e1 e2 : Exp) :
  m <> m' -> e1 |_| m -> e2 |_| m' ->
  listeningState (nextEq e1 e2)
| listenPiName (m m' : Mode) (r : Distance) (e1 e2 e3 : Exp) :
  ~possiblyIncompatible m m' r -> e1 |_| m -> e2 |_| m' -> e3 |_| r ->
  listeningState (nextPincCheckN $([e1, e2, e3])$)
| listenPiBody (m m' : Mode) (r : Distance) (e1 e2 e3 : Exp) :
  ~possiblyIncompatible m m' r -> e1 |_| m -> e2 |_| m' -> e3 |_| r ->
  listeningState (nextPincCheck e1 e2 e3).
```

Remark 7.2 (Why not a Behavioural Characterisation of State Predicates?)**.** It is conceivable that since these state predicates are defined in the name of abstraction, a behavioural characterisation would be more apt. In other words, rather than examine the syntax of processes, simply observe the actions available to a process in order to define its state. This would resonate with standard methods of process equivalence e.g. bisimulation already well established in the literature. Perhaps the development of such predicates, or even equivalences would be fruitful, and may even provide a more concise path towards the final proof. However, it is not clear how this would be so. It seems that there needs to be some reference to the actual process structure in order to formulate certain properties e.g. back tracking and forward tracking i.e. asserting which processes preceded or succeeded a given process over a transition. Hence the state predicates are defined as they are, chiefly looking at the syntax- along with possibly some other relationships among the parameters. ▲

Not all of the state predicates match the process definitions one-to-one. Some are more "high level". These are built from lower level state predicates and are hence referred to as compound or composite state predicates. Another less frequent term for these would be "umbrella" predicates. The constructors of such a predicate correspond to the lower level predicates that constitute it, which will be referred to as "contributory" predicates. Listing 7.22 gives an example of a compound state predicate. The idea of this predicate is to capture what it means to be a broadcast process. There are three contributory predicates for this, and a constructor for each, corresponding to the three lower level predicates outlined in Listing 7.16. These higher level predicates are useful in stating general properties such as "A broadcast component can never change the mode state".

Listing 7.22: Broadcast State Predicate.

```
Inductive broadcastState (p : ProcTerm) : Prop :=
  | broadBcwSt (m : Mode) (x : Time) : bcWaitState m x p -> broadcastState p
  | broadSlpSt : sleepingState p -> broadcastState p
  | broadBcrSt (m : Mode) (l : Position) : bcReadyState m l p -> broadcastState p
    .
```

A lifting function is necessary to transform properties about the broadcast component into properties about the overall protocol process. This is given in Listing 7.23. The function takes as an argument a proof that some property X holds for the process p and asserts that the broadcast-lifted X holds for the parallel composition of p with another two processes. There are similar functions for the overlap and listener components. This function is used to state properties such as "The broadcast component of the protocol process is in state S". While it is chiefly used to lift state predicate properties, there is no such restriction in the definition; indeed, the variable X may be any predicate on process terms.

Listing 7.23: Broadcast Lifting.

```
Inductive liftBroadcast (X : ProcTerm -> Prop) : ProcTerm -> Prop :=
  | lftbc (p p2 p3 : ProcTerm) : X p -> liftBroadcast X (p $||$ p2 $||$ p3) .
```

7.4.6 Specialised Tactics

Here a number of specialised tactics are discussed. These differ from the generic tactics of Section 7.4.2 in that they are geared specifically towards the structures making up the Comhordú model, and are thus of little use to the wider Coq community. A typical example of such a tactic is shown in Listing 7.24. This tactic is designed to destruct and entity into its constituent components and then further destruct the interface component of that entity. This would normally have to be done with two applications of the "destruct" tactic along with a lot of renaming and tidying. However, since it is such a common occurrence, this work has been amalgamated into a single tactic. The first argument to this tactic is the entity to be destructed. The remaining arguments constitute the names that will be given to the sub-components once the destruction has taken place, ending in the names of each of the lists comprising the interface component. There are many similar tactics to this one, for encapsulating the work of destructing of unfolding various structures in order to save space and time. This is a typical example of such a tactic and so is chosen here to represent this type of tactic.

Listing 7.24: Destruct an Entity into its various components and then further destruct the interface component.

```
Ltac destr_ent_inter e p l k li lo ln := let h := fresh in destruct e as [p l h k
    ];
  destruct h as [li lo ln].
```

Another typical type of specialised tactic that occurs in this model is what is referred to here as a "linking tactic". These tactics correspond to linking theorems, in which transitions at lower levels are shown to hold given that some higher level transition exists. In this case, for example, there is a theorem "link_ent_inter_del" which, given a delay transition between entities, yields a delay between the interface components of the entities in question. Applying this theorem in the context of a proof can be cumbersome, hence there is a tactic to do so. This tactic first matches against the entire proof context with the expression "match goal with". Following this match statement is a single clause after the vertical bar, though in general there can be many such clauses. The first part of the clause is the pattern to be matched. The pattern itself is broken into the hypotheses and the goal, separated by a turnstile "⊢". An underscore after the turnstile is a wild-card match, indicating here that the actual "shape" of the goal is irrelevant. To the left of the turnstile is a binding to a single hypothesis, which is locally named here as "H", but may be called anything in a given proof context. The type of H is a transition from one deconstructed entity to another, deconstructed in the sense that the components of the entity are explicit and visible. The tactic then applys the linking theorem to this hypothesis, producing a new hypothesis which is given a fresh name beginning with "U", a parameter that is passed to the tactic.

Listing 7.25: Link: Entity to Interface.

```
Ltac link_entinterdel_tac U :=
  match goal with
  | [ H : [|?p, ?l, ?h, ?k|] -ED- ?d ->> [|?p', ?l', ?h', ?k'|] |- _] =>
  let U1 := fresh U in lets U1 : link_ent_inter_del H
  end.
```

Depicted in Listing 7.26 is a tactic "taeAuto" that will automatically prove that certain processes are time action enabled i.e. capable of performing a timed transition. It does this by repeatedly drilling into the process until it finds either input or output prefixed terms, at which point the delay proof is simple. The tactic works by repeatedly applying a single step tactic until this is no longer possible; this repetition is specified with the "repeat" keyword. The single step tactic looks at the structure of a term and applies a previously proved result from there.

This tactic will only work for a certain class of processes that are, let's say "eventually discrete guarded sums" i.e. if they were put in some sort of normal form, with all the applications expanded to their definitions, then they would be sums of input or output prefixed terms. This is a good example of where automation via the tactic language is essential in Coq. A single application of this tactic is enough to show that various processes are time action enabled, whereas to attempt to do all those steps by hand would be very cumbersome.

Sometimes, auxiliary definitions in the Coq specification itself suffice instead of such a tactic. However, recursive functions in Coq using the "Fixpoint" operator are restricted by the rule that there must be a decreasing argument in every recursive call, an overly conservative condition that nonetheless guarantees termination. In this case, since the number of applications of the single step tactic is unknown in general, such a function in Coq would be impossible to define. Furthermore, tactic notation is often more intuitive and concise. The only drawback is that tactics are not guaranteed to work; however this is simply remedied by applying the tactic in the intended context: if it doesn't work then it can simply be discarded or changed. In this case, the tactic is successful, proving the result "delApp", which states that every application term can delay.

Listing 7.26: Timed Action Enabled Automation.

```
(** A single step of the full automation tactic for "eventually discrete guarded
    sum" terms.*)
Ltac taeAutoSing := match goal with
  | [ |- timedActEnabled ?P _ ] =>
    match P with
    | _ $< _ >$ ? _  => apply taeIn
    | _ $< _ >$ ! _  => apply taeOut
    | _ >> _  => apply taeThenBranch
    | _ $+$ _  => apply taeSum
    | _ $(_)$  => apply taeAppAuto; simpl
    | $< _ >$ _  => apply taeDel
    end
  end.

Ltac taeAuto := repeat taeAutoSing.
```

7.5 Pending Work

A number of results in the Coq model lack proofs. These results can be split into two categories: admitted theorems and conjectures. The former category are theorems are stated or partially completed, but could not be fully verified because of limitations of time. The latter are results that are assumed because their proofs are deemed obvious. There is no clear distinction between the two, and some of the results from one category could arguably belong to the other.

In this section, some of these results are presented. The results are only a representative sample since there are too many to list here. For the entire list of results, the reader is referred to the complete Coq model and to search for the keywords **Admitted** and **Conjecture**. Nonetheless, the sample chosen should cover most "categories" of results that are pending. This is achieved by selecting single results that in some way typify a particular type of result. Admittedly the categorisation is in places arbitrary, and indeed some results cannot be easily categorised. Also selected here are any results that seem interesting or whose solution appears to be non-trivial. Those that closely resemble others already presented are omitted here.

Along with each result presented here is a discussion of a proposed solution towards a proof of that result. Sometimes such a solution entails a general tactic that could apply to an entire category of results, as represented by the chosen example result. Together, these solutions should form an idea of how the remaining work on the Coq model would best be tackled, a possible future venture continuing from this work. To this end, there are also proof sketches, in varying degrees of detail, given as comments along with every admitted result and conjecture in the complete Coq model. These comments are included in most of the following listings.

Listing 7.27 is a typical example of an "inter-component" property i.e. a relationship between the various components making up an entity. In this case, the theorem says that for a reachable entity, whenever the predicate *switchCurr* holds for the software component of the entity with mode parameter m, then the next

mode of the entity is *m*. The proof would build upon similar results about states that must have preceded *switchCurr* e.g. *switchBc*. There would also need to be some results about when the mode state can/cannot be changed.

Listing 7.27: If we're in the state switchCurr m', then the next mode is m'. Corollary: switchCurr is urgent.

```
Theorem switchCurr_next_Mode (e : Entity) (m : Mode) :
  reachableEnt e -> switchCurrStateEnt e -> nextModeEnt m e. Admitted.
(**Proof: Relies on a chain of support of similar results for switchBc, ovWait,
    ovReady and init. Each one relying on the last. The ovWait and ovReady ones
    would need to be proved by mutual induction.*)
```

An urgency result asserts that time cannot pass for a certain component in a certain context. In the case of Listing 7.28, urgency is stated in terms of a reachable entity, given that the predicate *tfsListen* holds for that entity. While urgency could be stated at lower-level layers of the language e.g. the software layer or the mode state layer, the urgency results used in this proof development are generally at the entity level. The reason for this is that most of the time, in order to demonstrate urgency, a synchronisation must be shown to be possible between two entity components e.g. the software component and the mode state component. It should be possible to at least partially automate the proofs for this class of results via some suite of "urgency tactics" which reduce the proof obligation to demonstrating that some synchronisation is possible. It is possible that different tactics would exist for different types of synchronisation. For example, there might be a tactic that reduces the goal of urgency to showing that the software component itself is urgent. This in turn could be proved with a software-level urgency tactic.

In this case, it is indeed the software component alone that yields the urgency property. The state *tfsListen* means that the overlap component is ready to output on the channel *cUnpause*. It can also input on *cAbort* and *cBad*. Elsewhere it is possible to show that the listener component is always either urgent, capable of input on *cUnpause*, or capable of output on the channel *cAbort* or *cBad*. Hence urgency arises either directly from the listener component or as a result of a synchronisation between the listener component and the overlap component.

Listing 7.28: The state tfsListen is urgent i.e. time cannot pass for an entity in this state.

```
Lemma tfsListen_urgent (e e' : Entity) (d : Delay) :
  reachableEnt e -> tfsListenStateEnt e  -> e -ED- d ->> e'
  -> False. Admitted.
(**Proof: del_listening_pre gives the only states possible in which the listener
    component can delay. It can be shown that in all of these states, a
    synchronisation is possible with the overlap component which is in
    tfsListenState.*)
```

The previous result given in Listing 7.28 relies on an elimination result given in Listing 7.29. This results says that whenever delay is possible, the listener component can only be in one of four states. This is a succinct but equivalent way of stating that all *other* listening states are urgent. By proving the latter, and then performing a brute force check on all the cases for the state of the listener component, these four cases should be the only ones remaining. A similar result, not included here, exists for the overlap component.

Listing 7.29: A reachable entity that can delay is in the listening state.

```
Theorem del_listening_pre (e e' : Entity) (d : Delay) :
  reachableEnt e -> e -ED- d ->> e' ->
  abortOvlpStateEnt e \/ badOvlpStateEnt e \/ listeningStateEnt e \/
    pausedStateEnt e. Admitted.
(**Proof: (del_listening): Since the entity is reachable, we know it has a
    listener component (...reachableProt_triple...), so we do a case analysis as
    to what the location of the listener component is. In most cases, we can show
```

```
    by simple urgency results that a discrete action is possible via the
    listener synchronising with another component of the entity. The remaining
    cases that aren't urgent are exactly the conclusion to this result, and we're
    done.*)
```

Many of the remaining results to be proved are simple "lifting" results. An example of such a result is given in Listing 7.30. This result says that whenever the current mode of an entity is *m* based on the predicate currModeEnt at the entity level, then it is also *m* according to the network level predicate currModeNet. This result is really quite trivial, which is why it is left as a conjecture, but it has been included here simply to demonstrate the nature of a typical "lifting result". In general, such a result takes a predicate defined at one level and lifts it to the next level, in this case from the entity to the network level.

Listing 7.30: Lifting result: from the entity level to the network level.

```
Conjecture currMode_intro : forall (m : Mode) (e : Entity) (i : nat) (n : Network
    ),
  m = currModeEnt e -> e @ i .: n -> currModeNet m i n.
(**Proof: Obvious- lift currModeEnt to currModeNet*)
```

A number of "time split" results exist across the various layers of the language. For any language with delay transitions in its semantics, this result states that a delay transition can be split into two smaller delays adding up to the original delay, with an intermediate state reached between these two smaller delays. An example of such a result is given in Listing 7.31, which states the time-split property for the entity language.

Listing 7.31: Every delay from e to e" can be split into two delays with some intermediate entity e'.

```
Lemma timeSplit_ent e e'' d d' d'' :
  e -ED- d'' ->> e'' -> d'' = d +d+ d' ->
  exists e', e -ED- d ->> e' /\ e' -ED- d' ->> e''. Admitted.
(**Proof: This should follow from time split properties of the underlying
    components. Also, it would have to be shown that the noSynch predicate would
    still hold for the intermediate entity component e', with the time parameter
    changed.*)
```

A linking result takes as a premise a transition at a certain language layer and uses this to infer various cases for transitions at lower layers of the language. For example, the result given in Listing 7.32 states that whenever a discrete entity transition is possible, then either the interface components of the source and derivative entities are equal or the mode states are equal and one of a number of possible transitions given holds for the interface component. Results like this are fairly straightforward but they are vital building blocks in the overall proof development.

Listing 7.32: Elimination theorem giving the possible cases for the interface component of an entity given that the entity has performed a transition.

```
Lemma link_ent_inter_disc p p' l l' h h' k k' a :
  [|p, l, h, k|] -EA- a ->> [|p', l', h', k'|] ->
  h = h' \/
  (k = k' /\
  ((exists v, (h -i- chanOutProc {? v -i> h' /\ p -PA- chanOutProc;! v -PA> p'))
    \/
  (exists v, (h -i- chanInProc {! v -i> h' /\ p -PA- chanInProc;? v -PA> p')) \/
  (exists v, (h -i- chanAN {! v -i> h' /\ p -PA- chanAN;? v -PA> p')) \/
  (exists v, h -i- chanIOEnv {? v -i> h') \/
  (exists v r, h -i- chanIOEnv _! v !_ r -i> h'))). Admitted.
(**Proof: Follows from the semantics of entity transitions. Destruct the
    transition in question and in every case one of the disjuncts of the goal
    will hold.*)
```

There are a number of results pending relating to the four path predicates that are defined over the listener process. To briefly recap from Definition 6.4, these predicates relate to the state of the listener when it is processing either an incoming message or a coverage notification. Given in Listing 7.33 is a result stating that an incoming message which is possibly incompatible with the next mode is either preserved by a discrete action or else the message abort path holds after that action takes place. Results like this one eventually build up to higher level results about reactions to events, which are essential parts of the proof.

Listing 7.33: If an incomingNet message is incompatible with the next mode and the position of an entity, then in the next state, it is either still incomingNet or the entity has entered the msgAbortPath.

```
Theorem incomp_incomingNet_abort (n n' : Network) (a : ActDiscNet) (l l' :
    Position) (i : nat)
  (m m' : Mode) : n -NA- a -NA> n' -> inPosNet l i n -> nextModeNet m i n ->
  incomingNet [baseMode m', basePosition l'] i n ->
  dist l l' - speedMax*msgLatency < possIncDist m m' ->
  incomingNet [baseMode m', basePosition l'] i n' \/ msgAbortPathNet i n'.
  Admitted.
```

Building upon the results about path predicates are results about reactions to events. These state that some event is always followed by a reaction. In Listing 7.34, a result is given stating that the delivery of a message with insufficient coverage exactly *adaptNotif* units in the past is followed by either a pending notification of said coverage, or the listener is processing this notification on the bad path, or a transition to a fail-safe mode has been initiated. Given the urgency of the first two cases, this result entails that once *adaptNotif* time units have been exceeded since the message delivery, a fail-safe transition is guaranteed to have been initiated. This is a key high-level result towards the safety of the system: insufficient messages cause transitions to fail-safe modes.

Listing 7.34: Cases for an insufficient message delivery that happened exactly adaptNotif time units ago.

```
Theorem instant_insuff_bad (n : Network) (m : Mode) (l l0 : Position)
  (r : Distance) (t : Time) (i : nat) (p : reachableNet n) :
  delivered ([-[baseMode m, basePosition l], l0, r-]) adaptNotif i n p -> ~
    sufficient m r ->
  currSince m t i n p -> adaptNotif < t ->
  incomingNetNotif [baseDistance r, baseMode m, basePosition l] zeroTime i n
  \/ notifBadPathNet i n \/ tfs m zeroTime i n p.
```

Some results are of a form relating the *currSince* relation to lower level state predicates. The result given in Listing 7.35 says that if the predicate tfs holds in one state and then not in the next, the *currSince* relation cannot hold in the first state. This entails that when a transition to a fail-safe mode is completed, the *currSince* relation ceases to hold. A result in the opposite direction is shown in Listing 7.36. Here, the holding of the *currSince* relation before a delay is shown to imply that the broadcast component is in the wait state with a time parameter at least equal to that delay.

Listing 7.35: If we are switching from a tfs state to a non-tfs state, then we cannot be currSince.

```
Lemma tfsState_not_currSince (m : Mode) (t : Time) (i : nat) (n n' : Network)
  (p : reachableNet n) (a : ActDiscNet) (w : n -NA- a -NA> n') :
  tfsStateNet i n -> ~tfsStateNet i n' -> ~currSince m t i n p. Admitted.
```

Listing 7.36: If we're currSince and we can delay, then we're in the bcWait state.

```
Corollary currSince_del_bcWait (n n' : Network) (m : Mode) (t : Time)
  (i : nat) (d : Delay) (p : reachableNet n) :
```

```
  n –ND– d –ND> n' –> currSince m t i n p –>
  exists x, bcWaitStateNet m (delToTime (x +dt+ d)) i n. Admitted.
(**Proof: Apply currSince_bc_state and then eliminate bcReady as it is urgent*)
```

The theorem in Listing 7.37 is in fact two conceptually separate but dependent results conjoined together for ease of proof by induction. Ideally these would be separate results proved by mutual induction, but this is complicated to achieve in Coq. This theorem talks about the relationships between time parameters of the *nextSince* relation and various state predicates. The first half of this theorem says that the time remaining in the state *ovWait* plus the time of the *nextSince* relation is always constant, equal to the wait time from the current to the next mode. The second half of the theorem says that the sum of the time parameters of *ovReady* and *nextSince* are always constant, equal to the wait time plus the period. Together, these time relationships guarantee a lower bound on the *currSince* relation because when it is entered the wait time parameter of *ovWait* must be zero and so the parameter to *nextSince* must be the wait time, which carries over to *currSince*.

Listing 7.37: Relationship between time parameters of nextSince relation and ovReady and ovWait states.

```
Theorem nextSince_ovWait_ovReady_time (n : Network) (m : Mode) (i : nat)
  (t u x y : Time) (l : Position) (p : reachableNet n) :
  (nextSince m t i n p –> ovWaitStateNet m u x y i n –>
  exists m0 q, addTime t x = waitTime m0 m q
  /\ addTime u y = addTime x (period m)) /\ (*Half theorem*)
  (nextSince m t i n p –> ovReadyStateNet m u l i n –>
  exists m0 q,
  addTime t u = addTime (waitTime m0 m q) (period m)). Admitted.
```

Some of the remaining results to be proved can be categorised as "high level". While there is no formal definition of what high level is meant to mean here, roughly these results are those whose statement involves auxiliary relations at the network level. Most of these results can be proved by induction on one of the auxiliary relations involved. Three examples of such results are given in Listing 7.38, Listing 7.39 and Listing 7.40. The explanations of these results are given in the captions, while proof ideas follow as comments.

Listing 7.38: If an entity is nextSince m t, then <m, l0> was sent exactly t time units ago for some l0.

```
Theorem fst_sent (n : Network) (m : Mode) (t : Time) (i : nat)
  (p : reachableNet n) : nextSince m t i n p –>
  exists l0, sent [baseMode m, basePosition l0] t i n p. Admitted.
(**Proof: By induction on nextSince. In the base case we have that the previous
    state was ovReady m t l0 for some l0. This, coupled with (...ovWait_prev...)
    allows us to assert that the action linking the states is then a tau action
    which decomposes into an output by the overlap component of the entity i on
    io of <m, l0>. We use (...linking...) and (...prot_io_in_not...) to assert
    that this action is complemented by an input of same by the interface of that
     entity. Which gives us the base case for sent, with t = 0. Hence we use l0
    as our existential witness and satisfy the goal. In the inductive cases, we
    simply  apply the inductive hypothesis and then the appropriate constructor
    of sent.*)
```

Listing 7.39: If an entity has sent a message v mL time units ago, then said message is in the output queue of the interface component with timestamp 0 or it has been delivered 0 time units ago.

```
Theorem sent_out_del (n : Network) (v : list Base) (i : nat)
  (p : reachableNet n) : sent v msgLatency i n p –>
  outgoing v zeroTime i n \/
  exists r l, delivered ([–v, l, r–]) zeroTime i n p.
  Admitted.
```

```
(**Proof: By induction on sent. The base case fails because it gives the
    contradiction mL = 0, while mL is assumed positive. For the discrete
    inductive case we get from the I.H. that either outgoing v 0 i n or
    delivered v r 0 i n. If the latter is true, then we can immediately show by
    constructor that delivered v r 0 i n'. Otherwise we know outgoing v 0 i n.
    In which case we can apply (Basics::outgoing_timeout_disc) to obtain our goal
    . Moving on to the timed case, where we get that t + d = mL and so t = mL - d
    < mL, and so by (sent_outgoing) we have outgoing v (mL - t) i n, which by
    some algebra and substituting for t using our recently deduced equality is
    the same as outgoing v d i n. Then by (Basics::outgoing_del) we get outgoing
    v 0 i n, as required.*)
```

Listing 7.40: If an entity is nextSince m for t time units and t is greater than the message latency, then for some l0 it has delivered a message <m, l0> to some r at t - mL time units ago, and the l0 in the message differs from the current position by at most Smax*t.

```
Theorem fst_delivered (n : Network) (m : Mode) (t : Time) (i : nat)
  (l : Position) (p : reachableNet n) :
  nextSince m t i n p -> inPosNet l i n -> forall q : msgLatency < t,
  exists l0 l1 r,
  delivered ([-[baseMode m, basePosition l0], l1, r-])
  (minusTime t msgLatency (RIt_le msgLatency t q)) i n p /\
  dist l l0 <= speedMax*t. Admitted.
(**Proof: We apply (fst_sent) to achieve sent <m, l0> r t i n p. The we apply (
    sent_pos_bound) to get that |l - l0| <= Smax*t. Taking this l0 as our
    existential witness then, it remains to show that for some r we have
    delivered <m, l0> r (t - mL) i n p, which immediately follows from an
    application of (sent_delivered).*)
```

The results given in Listing 7.41 and Listing 7.42 are examples of what can be called "brute force" results. That is, these are results that can be proved by exhaustive analysis of the possible state predicates that can hold and a process of elimination. Such a proof method should be easily automated with a number of tactics. In general, this type of result claims that only certain state predicates can hold given some premise. For example, in Listing 7.42, a broadcast state that has just output on the channel *chanOutP* is shown to be initially in the state *bcReady* and transitions to the state *bcWait*. The proof of this proceeds by case analysis on all the possible broadcast states and showing that only the aforementioned pair match the transition in question.

Listing 7.41: The only two possibilities for a change in mode state as per a discrete transition are the mode switch or the mode transition to fail-safe.

```
Lemma mState_switch_states : forall (n n' : Network) (i : nat)
  (a : ActDiscNet) (k k' : ModeState), reachableNet n -> k <> k' ->
  n -NA- a -NA> n' -> mState_in_net k i n -> mState_in_net k' i n' ->
  (tfsCurrStateNet i n /\ tfsBcStateNet i n') \/
  (switchCurrStateNet i n /\ switchListenStateNet i n'). Admitted.
(**Proof: The mState_in_net predicates can be taken apart to show that there's an
     entity in the network. Then because the network is reachable, the software
    component of this entity is a protocolState process. Also, it can be shown
    from the semantics of entities that the only law (what about init?) that
    changes the mode state is that in which the software component outputs on
    mCurr. Brute force analysis of the possible states of each 3 components of
    the software component show that the only matching cases for this are the two
     disjuncts. That is, the only capability a protocol process has of writing on
     mCurr is from the overlap component in those specific states.*)
```

Listing 7.42: The only case for an output on the channel chanOutProc for the broadcastState is a transition from bcReadyState to bcWaitState.

```
Theorem bc_outProc_out (p p' : ProcTerm) (m : Mode) (l : Position) :
  p -PA- chanOutProc ;! [baseMode m, basePosition l] -PA> p' -> broadcastState p
    ->
  bcReadyState m l p /\ bcWaitState m (period m) p'. Admitted.
(**Proof: Brute force try all the other possible broadcast states (auto tactic)
    and this is the only one that matches*)
```

The term "preservation result" applies to a result that involves some property or predicate which is preserved by one of the transition relations. Given in Listing 7.43 and Listing 7.44 are examples of discrete preservation theorems i.e. relations that are preserved by the discrete transition relation on networks. In Listing 7.45 a delay preservation result is shown. Delay preservation in fact applies to all of the state predicates: if a state predicate holds for a certain network, and this network evolves via a delay to some derivative network, then the same state predicate will still hold for the derivative. The only change that may occur is to time parameters, if they exist, of the state predicate in the derivative network.

Listing 7.43: The tfsState predicate is preserved as long as the current mode is not fail-safe.

```
Lemma tfs_currMode_disc_pres (n n' : Network) (a : ActDiscNet) (i : nat) (m :
    Mode) :
  n -NA- a -NA> n' -> currModeNet m i n -> ~ failSafe m ->
  tfsStateNet i n -> tfsStateNet i n'. Admitted.
(**Proof: by noticing that tfs can only be left by tfsListen, which in turn can
    be shown by (inter-component) result that the curr mode then must be fail-
    safe, hence contradiction.*)
```

Listing 7.44: If v is outgoing with non-zero timestamp t, and the network performs a discrete action, then v is still outgoing in the derivative network.

```
Theorem outgoing_disc_pres (n n' : Network) (a : ActDiscNet) (v : list Base) (t :
     Time) (i : nat) :
  n -NA- a -NA> n' -> outgoing v t i n -> 0 < t ->
  outgoing v t i n'. Admitted.
(**Proof: invert/destruct outgoing to [p, l, {li, lo, ln}, k] @ i.: n & v, t : lo
apply net-ent disc linking tactic
Case entities are equal: constructor & rewrite
else apply ent-interface disc linking tactic
Case interfaces are equal: constructor & rewrite
else invert the interface transition
For cases where lo is preserved try constructor & rewrite Which should leave just
     the lo that transitions, then apply disc_pres theorem for timed lists

...or...

invert/destruct outgoing to [p, l, {li, lo, ln}, k] @ i.: n & v, t : lo
apply net-ent disc linking tactic & destruct tactic on the derivative ent
then case analyse lo = lo' or lo <> lo'
Former case easy, latter case, further linking theorem showing lo -a- lo', and
    then from there have disc pres for timed lists.
*)
```

Listing 7.45: The delay relation preserves the nextSinceState predicate.

```
Theorem del_pres_nextSinceState (p p' : ProcTerm) (d : Delay) :
  nextSinceState p -> p -PD- d -PD> p' -> nextSinceState p'. Admitted.
(**Proof: Inversion on nextSinceState into all possible cases and apply
the simply delay preservation results on these.*)
```

Some of the results remaining to be proved may be referred to as "disjoint state" results. These results assert that certain pairs of state predicates cannot hold simultaneously. This is done by stating that the simultaneous occurrence of both state predicates leads to a contradiction. For example the result given in Listing 7.46 states that whenever a network is initial, it cannot be in the state *bcWait*. These results should be amenable to automation via tactics. The idea is to invert the state predicates involved to the point that a contradictory equality is reached as a hypothesis. There is also some case analysis that probably will need to be done as part of the tactic.

Listing 7.46: If a network is initial then bcWaitState does not hold.

```
Lemma initial_not_bcWait_net (n : Network) (m : Mode) (x : Time) (i : nat) :
  initialNet n -> bcWaitStateNet m x i n -> False. Admitted.
(**Proof: Contradiction follows directly from the fact that initial states are
    running the protocol process, which has its overlap component in the dormant
    state. which is in turn disjoint from bcWaitState.*)
```

A good deal of the reasoning in the proof development involves analysing various transitions to/from certain state predicates and narrowing down the possible forms of these transitions in terms of both the action involved and the source/derivative state. These results are referred to as "tracking results" because the transition is tracked either backwards or forwards in order to gain more information. An example of such a result is the theorem presented in Listing 7.47, which says that whenever the state predicate *ovWait* holds for a derivative state, then either the predicate *init* holds for the source state, or the predicate *ovReady* holds, with various constraints on its parameters, and the action linking the states is an output on the channel *chanOutP*. It should be possible to develop tactics for the automatic treatment of such results. These tactics would most likely unfold the state predicates, analyse the body and determine which actions are possible, and then match these to the goal.

Listing 7.47: The predicate ovWaitState is preceded by either initState or ovReadyState.

```
Theorem ovWait_prev (p p' : ProcTerm) (a : DiscAct) (m : Mode) (t x y : Time) :
  p -PA- a -PA> p' -> ovWaitState m t x y p' ->
  initStateProt m p \/ exists l,
  ovReadyState m (addTime t (period m)) l p /\ x = t
  /\ y = period m /\ a = (chanOutProc ;! [baseMode m, basePosition l]). Admitted.
```

Similar to tracking results are what are called "activation results". These assert that a certain state predicate is capable of doing a certain action i.e. it is activated on that action. There can also be negative activation results, stating that an action is not possible in a certain state. One of these negative results is given in Listing 7.48, stating that the listener state is incapable of output on the channel *chanOutP*. In a similar manner to tracking, it should be possible to develop tactics to automatically build the proofs of these results.

Listing 7.48: A listener process cannot output on the channel outProc.

```
Theorem listener_outProc_out_not (p p' : ProcTerm) (v : list Base) :
  p -PA- chanOutProc ;! v -PA> p' -> listenerState p -> False. Admitted.
(**Proof sketch: Because none of its constituents can output on said channel.*)
(**Proof: tracking result*)
```

To end this section, let us present some statistics of the remaining work to be done in the Coq files. Table 7.1 gives a breakdown of the pending work for each file in terms of the number of admitted results, conjectures

and parameters. Also the totals across all files in each of these categories are given in the final row. While these numbers may seem large, many of the results are similar in nature and hence should be amenable to (semi-)automatic proofs using general tactics.

Table 7.1: Breakdown of Pending Work across Files.

Filename	Admitted	Conjecture	Parameter
EntAuxDefs			4
EntAuxResults	19		
EntityLanguage	2	2	1
InterfaceLanguage	4	3	1
Main			1
ModeStateLanguage	1		
NARIncomp	3		3
NARInsuff	5		1
NARMisc	14	1	6
NARMsgPosition	3		2
NARNonFS_currSince	5		2
NAROvlpTime	6		5
NARTop	5		3
NetAuxBasics	29	26	5
NetAuxDefs			8
NetworkLanguage	3		
ProtAuxResults	24		
SoftwareLanguage			7
Total	123	32	49

Chapter 8

Conclusions, Related Work and Future Work

This chapter concludes by summarising and discussing the preceding chapters and then exploring related work and possible directions for future work. To begin with, a summary of the contribution of this thesis is given in Section 8.1. A discussion of various issues relating to this work is formed in Section 8.2. The shortfalls of this work are considered separately to this discussion in Section 8.3. In Section 8.4, related work to formalising Comhordú is explored. Section 8.5 proposes a number of extensions to this work and examines a number of avenues for future work.

8.1 Summary of Contribution

This section abbreviates the content of the thesis so far. To recap, the goal of this thesis is to formalise and verify a pre-existing coordination protocol called Comhordú. A motivating factor for undertaking this formalisation is that the formal specification of a system demands precision as opposed to an informal specification, which may be ambiguous or lacking in detail. Furthermore, formal specifications are amenable to rigorous verification i.e. properties can be proved about the specifications using mathematically sound techniques. In summary, the contribution of this thesis consists of the following:

- A formal specification language for mobile distributed systems with wireless communication, which is revised in more detail in Section 8.1.1.
- A specification of the Comhordú protocol in the above language, summarised in Section 8.1.2.
- A proof of the correctness of the Comhordú protocol relative to a safety property, which is revisited in Section 8.1.3.
- An encoding of the previously mentioned language, protocol and proof in Coq, reviewed in Section 8.1.4.

The following subsections expand on the above points.

8.1.1 Language

Chapter 4 defines a language for the specification of systems of mobile agents. The language is referred to as multi-tiered or multi-tiered, meaning that it is composed of a number of sub-languages. These sub-languages can be categorised as either modelling external behaviour of entities or internal behaviour. External entity behaviour is embodied by the entity and network languages. The main features of these languages is message broadcast and entity mobility. Internal entity behaviour consists of the interaction of sub-components within an entity across channels. The internal components of an entity, which may be recalled from Figure 4.1, consist

of a software process, an interface modelling incoming/outgoing messages and a "mode state" component that keeps track of the mode of operation of the entity and allows for mode changes to occur. Each sub-component is modelled by a separate sub-language. The following list outlines the sub-languages.

- Section 4.1 The software language allows a piece of software to be specified for controlling an entity. The software component may communicate with the mode state and the interface component, as well as read the current position of an entity. The language is similar to the classical process algebra CCS with some modifications involving time, parameters, value passing and recursion. This fragment of the language is used to specify the Comhordú protocol.

- The interface language is formulated in Section 4.2. It models the transition of messages through the wireless network medium. When a message is sent by the software component of an entity, it does not immediately reach neighbouring entities, as is the case in reality. To model this latency in message transit, the interface component buffers outgoing messages leaving an entity, adding to each message a time stamp that must elapse before the message can be broadcast. The interface component also buffers incoming messages, though there is no time stamp on these. They are immediately available for consumption by the software component. Another feature of the interface component is the modelling of coverage notification messages. That is, every time a message is delivered by an entity to a certain coverage, the sending entity is notified that coverage within a bounded time. This models the behaviour of an underlying model called the Space-Elastic Model (SEM).

- Section 4.3 The mode state language models the mode of an entity and also models mode transitions. A term of this language, i.e. a mode state, is either a single mode or a tuple containing a source mode, a target mode and a time stamp. This tuple represents a mode transition from one mode to another. The time stamp represents the amount of time remaining until that transition becomes available.

- In Section 4.4 an entity language is constructed from the individual components previously mentioned. Internal computations of the entity correspond to communications between its various sub-components. Entities can also broadcast and receive messages through their interfaces. Mobility is modelled at this level by allowing an entity to update its position across a delay, where the magnitude of the change in position is bounded by an assumed maximum speed multiplied by the value of the delay. Once the entity language is defined, the development of a network language is straightforward, simply consisting of a system of entities operating concurrently and communicating via broadcasts.

Each of these languages is given a syntactic definition and a structural operational semantics, the latter being a set of rules governing the behaviour of terms in the language. All of the semantics incorporate the passage of time in their behaviour by means of delay transitions. These delay semantics are linked so that when a delay happens at the top level language, it propagates down to lower levels. Delays have only a minor effect on the structure of terms. The only things that change over the course of a delay are delay guards, time stamps and possibly the position of an entity. The latter behaviour is a novel feature of this language that generalises the notion of mobility bounded by a maximum speed. That is, entities are allowed to jump to anywhere within a bounded distance over a given delay, modelling movement but also allowing behaviours that clearly would not exist in the real world, as discussed in Remark 4.19, e.g. infinite acceleration. This inclusion of extra behaviours is not an issue as long as all desirable behaviours are included, which is the case.

8.1.2 Protocol

The Comhordú protocol is formally specified in Chapter 5. This formalisation solidifies the informal protocol description of Section 5.2. While this protocol builds upon ideas presented in [Bouroche, 2007], chiefly that of responsibility, the protocol specification itself is entirely novel to this work. To recap, the purpose of the protocol is to ensure safety within a system of entities, where safety is defined as entities never coming too close together based on their respective modes. To this end, each entity running the protocol operates by periodically

broadcasting messages to neighbouring entities whenever it is in a non-fail-safe mode. These neighbours are assumed to be also running the protocol.

The protocol also incorporates reaction to messages in its behaviour. Whenever an entity receives a message, the message is analysed for possible incompatibility and if such incompatibility is found, the entity begins to transition to a fail-safe mode. A fail-safe mode is one that cannot cause any incompatibilities within the system. Another feature of the protocol is reaction to coverage degradations. If the coverage for a particular sender entity degrades, it can no longer assume that its neighbours are receiving its broadcasts and so must immediately begin transitioning to a fail-safe mode. Entities are notified of these coverage degradations by an underlying lower-level model called the Space-Elastic Model (SEM) upon which Comhordú is based.

The formal protocol component consists of three sub-components composed concurrently: the broadcast component, the overlap component and the listener component. The functions of each of these are now discussed.

- The broadcast component is responsible for periodic broadcasts of the current mode of operation to neighbouring entities.
- The overlap component is responsible for initiating, completing and aborting mode switches, including mode switches to a fail-safe mode when this is deemed necessary.
- The listener component receives messages from neighbouring entities and coverage notifications. It analyses these messages and notifications and determines when a fail-safe transition or the cancellation of a mode switch is necessary. In either case, it notifies the overlap component accordingly, which in turn carries out the required action.

Each of these components is defined using the software fragment of the overall language presented in Section 4.1. The formal specification of the protocol components is itself a contribution in that their behaviours are precise, in contrast to the preceding informal natural-language description which is vulnerable to potential ambiguity and misinterpretation.

8.1.3 Proof

The purpose of the Comhordú protocol is to ensure safety in some sense for certain systems. Safety states that all pairs of entities are sufficiently separated according to their respective modes. The exact meaning of this separation relies on an auxiliary uninterpreted function. The proof of safety is exposed in Chapter 6. It builds upon a number of lower-level results.

A key component of the proof is the introduction of what are referred to as “history relations”. These are relations upon network traces that “remember” the history of the trace in question, so to speak. Combining various individual results about these history relations leads to a proof of safety. The individual results capture behaviours such as reaction to an incoming message, or the relationship between various entity components, to mention just a few.

Some results are given full proofs, whereas others are accompanied only by “sketch” proofs. The lack of complete proofs for every result is a weakness of this work. It is hoped that eventually this can be achieved. In the meantime, the mixture between rigour and sketch proofs at least comprises an intuitive argument towards protocol correctness. In any case, the proof is now in a manageable state, broken into a number of sub-proofs, which each individually do not seem complicated to prove.

8.1.4 Coq Encoding

The aforementioned language, protocol and proof are encoded in the Coq proof assistant. The use of a proof assistant adds an extra degree of rigour in comparison to formalisations that are done purely “on paper”. The language and protocol definitions in the proof assistant are fairly straightforward. The proof, on the other hand, is rather complex, consisting of a large number of results spread across various different files. Some of these

results are not proved, but instead are assumed. Each such unproved result introduces potential weakness into the model in that there is no guarantee that the result holds beyond an initial intuition that this is the case. Nonetheless, every such result is accompanied by a comment providing a sketch of how it should be proved. Furthermore, every result is supplemented by an explanation of its statement for human readability.

8.2 Discussion

A discussion of this thesis is now presented. The chosen methodology is justified in Section 8.2.1 i.e. arguments are made in favour of the use of a process algebraic system specification coupled with a machine encoding of this specification. Following this, the issues of validation and ambiguity are explored in Section 8.2.2. These issues arise when the relationship between the formal model and the original informal model is examined. Section 8.2.3 addresses the matter of the size of the codebase comprising the Coq model. Following this the implications of zeno behaviours, i.e. infinite behaviours over finite time intervals, are investigated in Section 8.2.4.

8.2.1 Justification for choice of Methodology

The methodology for formalising and verifying Comhordú entails using a custom-built language to specify the Comhordú protocol, proving by hand that this specification is correct with respect to a safety condition, and encoding the language, protocol and proof in the Coq proof assistant. This choice of methodology is supported/defended by the following points.

- There are a number of reasons for abandoning automatic analysis tools and instead deciding to use a proof assistant for formalisation. Firstly, doing so circumvents the state-space explosion problem: "A solution to avoid this [state-space explosion] is to use theorem provers" [Hilaire et al., 2010]. State-space explosion, i.e. the exponential increase in the number of states of a system relative to its size, pervades all fully automatic analysis tools. Furthermore, model checkers generally require concrete models for their analysis engines. That is the number of components e.g. entities, need to be specified and often specific parameter values of the system also need to be specified. Such constraints are too limiting for Comhordú, which should ideally be verified for an arbitrary number of entities and abstract parameters.

- The choice to use a custom built language as opposed to an existing language is largely due to the fact that none of the existing languages directly support all the necessary features for modelling Comhordú. Of course, it is plausible to assume that these features could be encoded by means of a "hack", but this seems like a complicated alternative to a custom-built language for the intended purpose. For example, having to emulate the properties of messages in transit using purely a software language leads to far more complex processes. This is because there is no upper bound to the amount of messages being sent at any time and so an unbounded number of software processes would need to be spawned i.e. created dynamically to model every message in transit.

  Now, the main advantage to using an existing language is that there are generally "sanity" results already proved about that language. However, encoding such a language into a proof assistant would require these results to be translated from hand-proofs into the specification language of the tool if they are to be used. This detracts somewhat from the advantage of a language with pre-existing results. The exceptions are the few languages that have been encoded into Coq, but there is even less choice among these than there is among the totality of existing languages. Thus, since the language and results need to be encoded into Coq anyway, there seems to be little downside to developing a new language tailored to the specific needs of the system that is to be specified. The upside, on the other hand, is a more concise, readable model and a separation of concerns at the language level between conceptually distinct features of the model.

- Arguably, there could be more pre-existing theories exploited in order to reduce the size of the proof. Perhaps there are even theories encoded into Coq that could be imported. Indeed, an original effort towards the development of the proof employed a real-time logic. Though the logic was encoded manually in Coq, the ideas were drawn from the literature on real-time logics, covered in Section 2.4. However this proved overly complicated and was abandoned in favour of the current approach using history relations. Some basic pre-existing libraries are imported as part of this model but most of the specification is done "from the ground up". Certainly it is likely that there alternatives to this approach which make better use of existing theories; along these lines some ideas for improving the proof structure are outlined in Section 8.5.

8.2.2 Validation

With the exception of purely theoretical pursuits, most formal models must be assessed in terms of their relevance to some informal blueprint or guidelines. This assessment is called validation, and is defined elsewhere as "determining whether the conceptual simulation model ... is an accurate representation of the system under study" [Kleijnen, 1995]. In this case, validation must take on a slightly different meaning, since there is no single "system under study"; rather Comhordú encompasses a wide range of systems. An alternative interpretation of validation might then be the investigation of whether the formalised version of Comhordú is an accurate representation of the informal model. This is still not quite satisfactory due to its inherent ambiguity: there are two informal models, the original informal model from [Bouroche, 2007] and the revised informal model presented in Section 3.3. In light of this, two interpretations for validation come to mind, as outlined in the following.

- How closely does the revised informal model resemble the original informal model?
- How well does the formal model represent the revised informal model?

Separate analysis of the relationship between the formal model and the original model does not seem like a sensible pursuit. It would likely reiterate many of the points made in the analysis between the revised and the original models.

The relationship between the revised model and the original model is already given extensive discussion in Section 3.4. To summarise that section, the revised model excludes various details from the original model that do not pertain to the safety condition. Furthermore, it adds an explicit draft for a behavioural model that is not present in the original model. Also, it draws upon the notion of responsibility from the original model, focusing on just one contract and expanding its definition to a precise set of behavioural guidelines for entities to follow. Overall, it can be said that the revised model removes several components from the original model and then develops the remaining components in more detail.

The formal model is intended to be as close to the revised model as possible. However, the one major difference is the level of detail and precision involved in the formal model. In particular, the behavioural model, which is only sketched out at the informal stage, is devoted an entire multi-tiered language in the formal model. This expansion of detail serves to precisely characterise any loosely defined concepts from the informal model. Thus the overall relationship from informal to formal model can be seen as one of disambiguation.

There is only one notable departure of the formal model from the informal model. This is the replacement of the idea of a contract with that of a protocol. The protocol is more rigorous than the contract and closer to the style of a programming language. It might be asked whether the conditions of the contract are emulated by the protocol. Intuitively, it can be argued that this is the case, e.g. periodic broadcast behaviour is demonstrated by the example trace in Section 5.3.1. However a formal proof that the protocol implies the contract conditions seems superfluous, requiring a separate formal statement of the contract. Such a pursuit would detract from the main concern of this work, which is to show that safety is maintained.

A related notion to validation, as it is defined here, is what might be referred to as fidelity to the real world. That is, it would be interesting to explore how faithfully does Comhordú characterise reality? It is stressed here

that such an exploration is beyond the scope of this work; the chief concern here is validation of the revised model against the original and validation of the formal model against the revised model. Still the issue of fidelity is worth highlighting here. In relation to this concept, the following question seems important: if a real-world system was built using the formal Comhordú framework, how applicable would the safety result be?

This question is inextricably linked to the notion of refinement, which roughly means adding detail to a specification to make it more concrete. For example in [Romberg and Grimm, 2004] a method is presented for mapping hybrid system specifications to C++ code. In the case of Comhordú then, can a real-world implementation of the formal model be developed such that the abstract conditions of this model still hold in the real-world setting? The achievement of this seems challenging, particularly with respect to the behavioural model characterised by the languages. For example it is difficult to imagine that a system could flawlessly provide the guarantees of the underlying Space-Elastic Model.

8.2.3 Size of Codebase

One important matter to address is the size of the codebase comprising the Coq model. The size to which this codebase would grow was unforeseen before Coq coding began. The experience of encoding the Comhordú model into Coq has revealed that this codebase exceeds original expectations in terms of its size. The reason for this is that even the simplest proofs in Coq in general consist of a significant amount of code, certainly more than would be intuitively expected by the unseasoned Coq user. Take for example a result stating that for any real number x, there exists some y and z, also real numbers, such that $x = y + z$. The proof of this simple result in the Coq model consists of 20 sequential tactic commands. Considering that such a result would generally be accepted without proof in any standard mathematical development, 20 tactics seems like quite a substantial size. It is this level of granularity demanded by Coq proof scripts that accounts for the fact that the model currently consists of roughly 10,000 lines of code, which exceeds original projections, particularly when it is noted that not all results are fully proved at this stage.

One question that may be raised is, can this verbosity be reduced? Indeed, there are a number of ways to do so. On the one hand, clever use of abstraction enables a programmer to prove reusable results. This is the case not just for coding in Coq, but for any methodology whatsoever. More specific to Coq is the possibility to assume results without proof. Such results can then be proved at a later stage. This is analogous to writing a "stub" method in a large program whose body is essentially empty so that higher level program constructs may be tested quickly; the exact details of the method can then be filled in at a later stage. However, unless assumed results are eventually proved they introduce vulnerability to the model in that they potentially compromise the soundness of the proof. Further attention is given to this matter in Section 8.3.

A more thorough means by which code verbosity can be reduced is through the use of automation. This involves writing tactics which replace human input. Often these tactics are custom-built for the application in question and are composed of a number of simpler tactics. The only real drawback to the development of tactics is time. It is hard to know when the advantages of developing a tactic will compensate for the time lost in doing so. As a rule of thumb, a tactic should only be written if there exists a possibility for its reuse, though it is difficult to know when this is the case a priori.

8.2.4 Implications of Zeno Behaviours

There is a thread of discussion and examples relating to the notion of zenoness running through this thesis so far. Zenoness for software processes is discussed in Section 4.1.8 and defined in Definition 4.11 while an example of a zeno process is given in Example 4.12. The issue of zenoness at the entity level is briefly discussed in Remark 4.21.

It is now time to address the implications of such behaviours. Recall that zeno behaviour is characterised by infinite actions performed in a finite time interval. At a first glance, this seems troublesome. However, after some consideration, it becomes apparent that zenoness in this model can be ignored. One reason for this is that

zeno behaviour is an artefact of the abstract process calculus and is not likely to happen in the real world; in fact it is impossible, since software actions always must take *some* time to execute. Moreover, safety is defined in terms of finite traces i.e. reachability proofs, and so zenoness cannot have any meaning in this context, because it is an inherently infinite phenomenon. Also, even with these considerations aside, zeno behaviours do not affect safety. A system may perform an infinite amount of discrete actions over a finite interval of time, but this does not imply a violation of safety as it is defined here. For these reasons, addressing the issue of zenoness is not given priority in this body of work.

Still, it is worth noting that zenoness may be an issue that would need to be tackled if properties beyond safety were to be explored. Solutions could aim to remove the possibility of zenoness altogether from the language semantics, or a focus could be placed on the design of a new protocol that exhibited no zeno behaviours. A mixture of both is also an option. An example solution would be to restrict the rule TEST in the language by adding some sort of timer that enforced a minimal delay between mode switch initiations, though at a first glance this seems somewhat restrictive.

8.3 Shortfalls

There exist shortcomings both in the methodology and contributions of this thesis. Some of these may have been avoided had different choices been made during the course of the work, but this realisation is only apparent now; such is the nature of hindsight. Many of the problematic issues raised here are addressed with solutions, some of which are suggested as future work in Section 8.5. The following sub-sections outline the main shortfalls of this work. Section 8.3.1 addresses the incompleteness of the Coq code and the issues raised by this. Section 8.3.2 explores the matter of the complexity of the language used to model Comhordú. Section 8.3.3 investigates the limitations of Comhordú as a candidate for application in real-world systems. A number of other limitations are outlined in Section 8.3.4.

8.3.1 Incompleteness of Coq Proof

The proof of safety given in Coq is incomplete. That is, a number of results comprising it are assumed rather than proved. Rather, informal sketch proofs in the form of comments accompany these results. The degree of detail in these informal sketch proofs varies among different results, though admittedly the proof in its current state cannot be accepted as mathematically rigorous. This lack of rigour of course gives rise to the possibility of human error. Potential solutions to reduce the level of this error are to supply more detailed, rigorous sketch proofs by hand or, ideally, to prove the theorems completely in Coq. One open issue is that there are likely some results which are incorrect and require modification. The extent to which this is the case is hard to gauge at this point; it will only become apparent when proofs of these results are attempted.

Still, a large number of results have been proved so far without much modification to the original proof layout- though there have been some minor changes made. Also the current state of the proof is a positive result in itself in that it links safety to various behaviours of entities. That is, while the proof does not filter right down to the level of the protocol behaviour implying safety, it does at the moment still demonstrate that certain medium level conditions imply safety. Thus any system satisfying these conditions is guaranteed to be safe, which itself can be considered a useful result.

8.3.2 Complexity of Language

The language used to specify the Comhordú model contains a large number of rules. One reason for this is that it is comprised of a number of different sub-languages, each with their own separate syntax and semantics definitions. Nonetheless, complexity seems like an inevitable feature of any language hoping to model a system with as many features as Comhordú e.g. mobility, coverage notification, broadcast.

A consequence of having many rules in these languages is that a great deal of effort is spent proving results that are intuitively obvious, but require induction and so consideration of all the possible syntactic or semantic cases. This raises the question as to why operationally characterised sub-languages are needed at all. That is, perhaps an alternative characterisation could be given for some components. For example, instead of separate sub-languages being defined for the mode state and interface components, results could simply be assumed about these, capturing their important behaviours. In other words, components could be modelled as "black boxes" with assumed conditions on their behaviour rather than the transparent structures with operational semantics used in the current approach.

On the other hand, choosing operational semantics for all the languages is a more homogeneous approach. While this is the current choice, it is probably not exploited enough for its homogeneity. For example, the fact that all rules follow similar patterns should allow for the development of generic tactics for use on any of the languages. However no such tactics currently exist, though they are mentioned as a possible avenue for future work in Section 8.5.

8.3.3 Applicability

One potentially problematic aspect of the formal Comhordú model is its applicability to real-world scenarios. There are a number of reasons for this. First of all, the abstractions made as part of the formalisation process exclude certain details of real-world behaviour, e.g. finite acceleration, and it may be that these details influence the behaviour of the system in unforeseen ways. Another way of looking at this is from the perspective of refinement: how can the abstract Comhordú model be refined to a real-world scenario such that results about the behaviour are preserved, or at least probably preserved? It seems a major difficulty in achieving such a refinement lies in implementing underlying components such as the mode state and the interface, the latter of which captures the behaviour of the Space-Elastic Model (SEM). Also, it is difficult to assess how useful the formal model will be for use in specific applications; this point is addressed further in Section 8.5.2.

An example of a potentially problematic abstraction pervading the entire language and model is the notion of an atomic action. That is, discrete actions are assumed to take no time. Most process calculi make this abstraction, but in reality actions will always take some time. This could have adverse effects on results e.g. the reaction of an entity to an incoming message in reality will not be instantaneous but will be delayed by the time taken to process the receipt of that message. To combat this particular issue, redundancy could be built into the model, at least at the level of implementation if not in the abstract model. For example, wait times could be extended and reactions to events could be made overly conservative. However in general it is non-trivial if not impossible to determine whether results from the abstract model will translate to a real-world application without that application actually being built.

8.3.4 Other Limitations

A number of other limitations to this work are listed below.

- The fact that entities always react to incompatibilities or coverage degradations by transitioning to a fail-safe mode could be criticised as overly conservative. The original model of contracts only requires that an entity transition to another mode that circumvents an incompatibility, but the mode does not need to be fail-safe. An initial idea towards realising this type of behaviour in the formal model is that entities could switch to another mode whose max-min distance of compatibility is less than the current one, rather than switching always to ta fail-safe mode. However, the details of this are not worked out here, and it is believed that to add such behaviour would significantly complicate the model.

- The implementation details of the underlying Space-Elastic Model (SEM) are omitted from this model. It is not obvious that the guarantees of SEM, e.g. coverage notification, can be provided by some concrete implementation of the model. This is further investigated in Section 8.5.1.

- The model might be criticised as somewhat simple. To defend this, on the one hand, brevity in a specification is conducive to its verification: larger models are significantly more difficult to verify e.g. the state-space explosion problem faced by model checking tools. Moreover, simplicity in this case is largely a prerequisite of generality: further inclusions of detail tend to reduce the range of systems to which the model is applicable. While generality is desired here, an alternative line of work would be to investigate more specific instances or classes of the model where behaviour is more constrained and so more detail can be included.

- The Comhordú model and protocol formalised in this work differ significantly from the original Comhordú model presented informally in [Bouroche, 2007]. This is the issue of validation covered in Section 8.2.2.

- Axioms in the model pose the threat of inconsistency. With axioms, there is no longer a guarantee of soundness. E.g. one could assume with the axiom keyword some proposition P and its negation $\neg P$, and derive false. Or something absurd could be assumed like $5 < 3$ and then false could be deduced. Or in a particularly contrived scenario, false itself could be assumed as an axiom. Intuitively, it does not seem that the axioms in this model introduce any such contradictions, but the possibility remains open. The only conceivable means of ruling out such inconsistency would be through meta-proofs about the model.

8.4 Related Work

Recall from Figure 2.1 that the research area of this thesis roughly falls in the intersection between formal methods and distributed systems. In Chapter 2 each of these two areas are discussed separately in order to prepare the reader for the ensuing material. Work that can be categorised as lying in the intersection between these two fields is omitted from that chapter because it is not relevant as background material necessary to understand the remainder of the thesis. Still, it is important to study this intersection because it is the most directly related to the work in this thesis. The following is a review of some papers that are deemed to be related to the formalisation of Comhordú. Section 8.4.1 presents some examples of formal theories that are mechanised using proof-assistant tools. Section 8.4.2 investigates related real-world applications of formal methods. A discussion of the related work presented follows in Section 8.4.3.

8.4.1 Mechanised Formalisms

There is a body of work in which formalisms are encoded into theorem provers and proof assistant tools. In this section, work pertaining to the tools Coq and Maude is considered. The relationship between such work and the formalisation of Comhordú in Coq is that the latter is done using a process algebra that is encoded into Coq. Hence studying work in which formal theories are encoded into tools provides a reference point against which the current methodology can be compared. This comparison can generate ideas such as alternative approaches or extensions to the existing approach.

The following papers are examples of formalisms that have been encoded using Coq.

- An encoding of the π-calculus in the calculus of (Co)inductive constructions is detailed in [Honsell et al., 2001]. Proofs of language properties are performed using Coq, including a theory of "strong late bisimilarity", which is basically an equivalence relation over language terms modelling when two terms have the same behaviour in some sense.

- Real numbers are given a constructive characterisation in Coq [Ciaffaglione and Honsell, 2003] and validated as sound against an axiomatisation. Also an imperative process calculus called "imp" is formalised in this paper. A good argument put forth here is that proof assistants like Coq are a good compromise between the creative freedom possessed by a human being and the rigour of machine validation. On the one hand, humans are prone to error whereas machines are notorious for making nonsensical decisions when trying to intelligently search state spaces.

- A two-level meta reasoning in Coq [Felty, 2002] stratifies the definitions of a logic over two layers. A "specification logic" is defined in Coq, on top of which "object logics" are defined. The basic idea from this paper seems like it could be used in an alternative encoding of the multi-tiered calculus for Comhordú. That is, splitting a language over two levels allows a lot of elements from the higher order language to be recycled, which is a feature of abstraction in general.

- "A certified Compiler for an Imperative Language" [Bertot, 1998] involves verifying the correctness of a compiler. The compiler converts simple imperative programs into simple machine code. The language is encoded in fragments i.e. using a number of interlinked types. A useful feature of Coq is exploited in this paper: the conversion algorithm of the compiler is extracted automatically as a program with an accompanying proof of its correctness. Future work proposed is to increase the complexity of the languages involved.

A number of other theories have been machine-encoded using the Maude theorem prover. These are as follows.

- A meta theory for structural operational semantics (SOS) is developed in [Mousavi and Reniers, 2006]. This allows for specific languages with SOS to be encoded using the general meta structures.

- A technique for implementing SOS in Maude is devised in [Verdejo and Martí-Oliet, 2006]. The technique involves the use of what are called rewrite rules to represent transitions in the SOS. Case studies supporting this technique using actual languages e.g. CCS are provided.

- In [Rosa-Velardo et al., 2006] an operational and static semantics for a "Typed Mobile Ambients" calculus is established in Maude. In conjunction with this a type system is devised.

- "An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude" [Thati et al., 2004] develops a machine-encoded version of asynchronous π-calculus in Maude using rewrite rules to represent the operational semantics transitions. Also explored in this paper is the theory of may-testing equivalence on processes.

8.4.2 Formalised Systems

The formalisation of certain real-world systems comprises research that is directly related to the formalisation of Comhordú. The type of systems to which Comhordú applies are those in which multiple agents operate concurrently, move about in space and communicate over a wireless network. Within the context of this class of systems, Comhordú is concerned with the problem of coordination. Work that is directly related to this thesis then involves the formalisation of such systems, particularly with the added theme of coordination. Since this description of systems is quite general, a wide range of disciplines are covered e.g. land-based vehicles, aerial vehicles, and trains, among others. The degree to which these systems are relevant varies from system to system e.g. some systems are more focused on the specifics of entity dynamics rather than coordination among entities.

There exists some work that overviews the application of formal methods in various areas. For example [Bertolino et al., 2012] details the challenges faced in validating and verifying autonomic systems. It also discusses some methodologies for tackling these challenges, and proposes a new methodology towards this end. The methodology is tested in a case study of the wine making process. Verification, validation and testing (VV&T) is the subject of [Khan et al., 2014]. The argument put forth is that VV&T is a necessary approach towards meeting the ever increasing demands for embedded systems in terms of cost, timeliness, customer satisfaction and quality. An industrial case study is used to illustrate this point.

The coordination of autonomous vehicles is a recent theme in the literature of growing popularity. Coordinated evasive manoeuvers are introduced in [Althoff et al., 2010], with the intention of achieving safety despite the threat of uncertain events e.g. a vehicle skidding on a road. Safety in this setting is verified by

computing the set of reachable states for a vehicle and ensuring that there is no overlap between these and any obstacle. A methodology for the verification of cooperating traffic agents is given in [Damm et al., 2007]. The methodology is demonstrated using a case study involving the European Train Control System (ETCS). In the ETCS example, trains react proactively to changes in environment by braking and stopping before they cause a safety violation. This happens when the train reaches the end of an authorised piece of track. This authorised piece of track moves dynamically in front of the train as the train ahead moves out of the way, allowing for smoother operation of the system than would be the case for a static division of the track into segments. This is quite similar to the slot-based driving approach of [Marinescu et al., 2010] already discussed in Section 2.2. Also discussed in that section is a formalisation of a protocol called CwoRIS [Asplund et al., 2012] but it is revisited here from a formal perspective. The purpose of the protocol is to achieve coordination among cars entering a junction. A constraint specification language called SMT-lib is used to specify the system, which is then verified using the Z3 theorem prover. A limitation of the model is that some cases are not considered in the verification. Also the detail and scale of the model could be enlarged.

Another related formalisation of a vehicular system is given in [Mitsch et al., 2012]. The system consists of vehicles on a motorway communicating via a central agent. Models of the vehicles' dynamics are formulated using the language of hybrid programs (HP). Safety of the system is then specified using a property specification language called differential dynamic logic. The tool KeYmaera is used to prove that the system is correct in terms of an absence of collisions between cars.

Aerial vehicles are another target for formal methods. Human-piloted aircraft are provided with "conflict resolution manoeuvres" in [Tomlin et al., 2001], which may be viewed as "rules of the road" for aeroplanes. The adoption of such manoeuvres leads to a decentralised system, and may reduce the bottleneck on air traffic control, increasing throughput. Safety in this case is stated in terms of protected zones about an aircraft never overlapping. This is similar to the idea of safety in Comhordú, which states that sufficient distance must be held between any two entities in a system, relative to their respective modes. In favour of formalisation vs. just testing, the paper argues: "it is not sufficient to simply simulate different conflict resolution techniques to illustrate their safety". A similar argument can be made in favour of formalising Comhordú. The formalisation of the aircraft is carried out by hand in this case.

A large amount of recent research exists in relation to unmanned aerial vehicles (UAVs). This is illustrated in the following.

- An application of formal methods to the verification of unmanned aircraft systems (UASs) is assessed in [Webster et al., 2011]. The correctness of such systems is defined in terms of the unmanned systems abiding by the same protocols as human pilots do. A simple UAS is modelled in the language PROMELA and verified using the tool SPIN. Then a more complex system is modelled and verified using an AI language called Gwendolen and the Java verifier AJPF. An interesting point is made in this paper: "Another possible difficulty [with formal modelling] is in justifying the abstractions made during the modelling process." This difficulty in justifying abstractions is a key challenge to overcome in any formal model because on the one hand, a system that is too specific is unwieldy to verify whereas on the other hand some details cannot be lost to abstraction.

- Collision avoidance for UAVs is the subject of [Peng et al., 2012]. The collision avoidance strategy consists of local optimisation and communication between the UAVs. In a similar vein to [Tomlin et al., 2001], vehicles are encircled with an imaginary protected zone and collision avoidance entails that no zones overlap.

- A slightly different approach is taken in [Carle et al., 2013], whereby UASs are modelled and assessed to deduce which event combinations lead to failures. This is a testing based approach rather than an exhaustive state-space search of possible behaviours. The modelling language used is UML, which is not formal, and the implementation is done in C++.

- In [Brunel and Cazin, 2012] the notion of a "safety case" is addressed. Such a case consists of some

concrete evidence combined with a demonstration that the evidence entails safety. A formal framework is developed for arguing that a system is safe, with the ultimate goal of automatic validation via tool support. The framework is demonstrated in a test scenario whereby a UAV is integrated into the air traffic.

Train systems are a popular subject for formal verification. The European Train Control System (ETCS) specification is subject to some formal analysis in [Platzer and Quesel, 2009]. A number of constraints on free parameters are identified as part of the analysis, constraints that are non-existent in the original specification. The Automatic Train Protection (ATP) project is the focus of [Xie et al., 2011]. The contribution of the paper is a formal analysis of Train-to-ground Communication Link Verification (TCLV). The formal analysis is achieved via the VDM-SL tool in conjunction with some manual proofs by hand. This resembles the formalisation of Comhordú presented in this thesis in the sense that proof by hand is combined with a machine-based approach. It is believed here that for a system of significant complexity, this is the most realistic approach.

Some applications of formal methods to the field of "multi-agent" systems (MASs) exist. MASs usually consist of a number of autonomously operating agents interacting together in some environment. A significant body of research in this area pertains to developing "intelligent" systems in some sense and as such there is an overlap between this field and the natural world. For example, a model of agents is given in [Niazi and Hussain, 2011] and demonstrated with a simulation of self-flocking animals. The formal language Z is used to model both the environment and the network of agents. Simulations of agent-based systems are also covered in [da Silva, 2012]. In this paper, agents are defined as having learning, drives and emotion and interact with their environment. Simulation involves evolving a system along a trace. The language Z is also used in this work for the purpose of formalisation.

8.4.3 Discussion of Related Work

Related work covered here falls into two broad categories: mechanised formalisms and formalised systems. The former relates to the current thesis in that the translation of a process algebra into the Coq proof assistant constitutes mechanising a formal specification language. The latter are related in that they are formalisations of real-world systems, while Comhordú is intended for real-world application. Because this literature is cross-disciplinary, it is difficult to classify it. For example, formalising a system of autonomous vehicles can be viewed from either the perspective of formal methods or distributed systems.

The mechanised formalisms presented here form a subset of the entire collection of formalisms available. None of these seem suitable for the purpose of modelling Comhordú. This is not surprising, since the demands on a modelling language for Comhordú are quite specific e.g. time, mobility and broadcast. In fact, even outside the scope of theorem provers and proof assistants, there doesn't seem to be any formalisms directly suitable to meet these requirements.

The presentation of formalised systems given here is a small sample demonstrating the type of existing work on the application of formal methods to real-world systems. It seems from this sample that the use of formal methods usually entails quite a narrow focus. That is, a specific algorithm, or scheme is usually tested and often demonstrated using a simple example. Also, systems are usually heavily abstracted. Abstraction is a necessity as is the case for any theory attempting to model the real world. It also facilitates simple system descriptions that are amenable to formal analysis. Similar abstractions exist in the case of Comhordú. For example, entities are modelled simply as points. Details of their weight, shape, temperature etc. are all omitted from the model.

8.5 Future Work and Possible Extensions

There are many opportunities for future work stemming from the current formalisation of Comhordú. The most pressing task is the completion of the pending work in the Coq model, outlined in Table 7.1. This pending

work chiefly consists of proving a number of theorems which are currently assumed without proof. After this, there are a number of future directions which this work might take. Section 8.5.1 addresses the matter of formalising and verifying an implementation of the Space-Elastic Model (SEM) underlying this model. Potential applications of Comhordú are explored in Section 8.5.2. A plethora of other ideas for future work are given in Section 8.5.3.

8.5.1 Formal Model of SEM Implementation

In this model of Comhordú, an underlying communication model is assumed called the Space-Elastic Model (SEM). However, its implementation details are omitted. Rather, a set of behavioural guarantees are simply assumed. The chief guarantee of SEM is that entities receive feedback on the state of the coverage in the wireless communication medium. It is natural to ask whether such guarantees are achievable in the real-world?

An implementation of SEM called the Space-Elastic Adaptive Routing protocol (SEAR) is proposed in [Hughes, 2006]. It would be interesting to model this in its own right and verify that the guarantees of SEM are indeed maintained by this implementation. It is likely that a contemporary process calculus for wireless networks such as CWS [Mezzetti and Sangiorgi, 2006] would be useful for modelling it, though this requires further investigation.

8.5.2 Potential Applications of Comhordú

The original Comhordú model [Bouroche, 2007] is an alternative approach to consensus for achieving coordination in systems of mobile entities. Intended applications of this original model are primarily traffic-based e.g. systems involving emergency vehicles and cars. However, the formal model differs from the original informal model, as discussed in Section 8.2.2. For the sake of brevity, the formal Comhordú model is henceforth referred to in the remainder of the section as simply Comhordú. In light of the revisions made to Comhordú with respect to the original model, the application domain of Comhordú must be considered in its own right. However, it is stressed here that overall, the primary focus of this work is formalisation, not application. This section presents some initial ideas for the application of Comhordú. To begin with, some "toy" example applications of Comhordú are illustrated, after which a small selection of literature is reviewed from the perspective of finding applications for Comhordú.

Remark 8.1 (Real-world Testing and Simulations)**.** In order to truly gain confidence in Comhordú beyond merely a theoretical model it will be necessary to explore applications in considerable detail, beyond merely some suggestive prose. This exploration could be carried out via computer simulations or even testing in a real-world system of entities e.g. a UAV team. However, for the time being, this is left as future work. ▲

The following are "toy" examples of applications that are envisaged for Comhordú. Further development of these ideas could potentially lead to real-world applications for the protocol, but for the moment these are just simple examples for demonstrative purposes.

- Consider a team of robots for collecting pieces of metal from the ground. Such a system could have many uses. For example, the robots could be employed to remove scattered sharp objects like shrapnel in a war-zone or nails and screws from the rubble of a building site. Imagine that the robots do so by the use of magnets. Robots have downward-facing magnets that attract the pieces of metal as the robots navigate the environment. The robots can then periodically dump all of the metal into an on-board container by lifting the magnet over the container and turning it off. For efficiency, it is imagined that these periodic "refills" are only done after a significant amount of metal has built up on the magnet.

  It is possible to say that robots have two "modes" in an abstract sense: **magnet-on** or **magnet-off**. The switch from **magnet-off** to **magnet-on** is instantaneous. However the switch from **magnet-on** to **magnet-off** requires any loose metal on the magnet to be dumped into the on-board container on the robot first. Robots can be assumed to move around more or less at random, though it is imaginable that

there would be high-level algorithms in operation to ensure coverage of the entire region e.g. divide and conquer. From the point of view of Comhordú these algorithms are disregarded and movement or position is not assumed to play a part in the mode of a robot.

Now, if two robots come too close together, let us say in this scenario that their magnets will attract each other and the robots will become stuck together. Furthermore, let's assume that the only way to separate the robots is for one of the magnets to be turned off. If both magnets have pieces of metal attached to them, then turning off a magnet would cause metal to be dropped. This is not desirable. Of course, the lost metal would probably fall into quite a concentrated region of space, making it relatively easy to gather once again, but let us say for the sake of the example that dropping metal constitutes unacceptable behaviour.

Here is where the Comhordú protocol can be applied. There are two modes in the system: **magnet-on** and **magnet-off**. The mode **magnet-off** is taken to be the fail-safe mode in the system. The mode transition time from **magnet-on** to **magnet-off** would be determined by the time it would take a robot to lift its magnet above its on-board container. Employing the Comhordú protocol in this scenario would ensure that no two robots with their magnets on would ever get stuck together because by the time they became close, one would have dumped all its metal pieces and turned its magnets off. Of course, wireless communication among robots is assumed here.

- Let us consider now a more safety-critical example in which fire-safety is a concern. Imagine a scenario in which a team of robots is employed for the job of welding for repairs. Perhaps the robots could be UAVs deployed to reach areas that are too high or risky for humans to visit. Consider for example a high suspension bridge or a skyscraper.

  In any case, let us imagine that there are two types of robots involved: welders and "finishers". The first type are robots that use a high-temperature flame or electric arc to weld an area of the structure in order to repair it. The second type of robot applies a finishing coat to the welded surface. This could be some sort of protective enamel, paint or even a coolant to solidify the welded material before it is damaged by atmospheric elements. The exact details are unimportant beyond a key property: the substance applied by the finishers is flammable at high temperatures. This means that a welder and a finisher cannot be in the same vicinity while both are active.

  There are four modes in this system: **welder-on**, **welder-off**, **finisher-on** and **finisher-off**. While it is probable that any system of this type would use distinct entities for welding and finishing respectively, it is still possible to imagine that some entities might be equipped with the capabilities of both welding and finishing. Irrespective of this detail, the fail-safe modes in the system are **finisher-off** and **welder-off**. The modes **finisher-on** and **welder-on** together pose a potential incompatibility, which would be captured by a non-zero minimum distance of compatibility being set for these modes. Potentially also two welders together could pose a threat to the system and so the combination of **welder-on** and **welder-on** may also have a non-zero minimum distance of compatibility.

  Now, employing the Comhordú protocol in this scenario would ensure safety in the sense that welders/finishers would switch off their on board components when in the vicinity of each other. A quirk of this model which makes it questionable in terms of real-world applicability is that a welder that is turned on cannot be incompatible with another robot whose welding/finishing component is turned off because these latter modes are defined as fail-safe. A possible interpretation of this is that when a device is switched off, the entity deploys a protective shield that is fire resistant.

- A form of collision avoidance is achievable with the Comhordú protocol. In comparison to more standard approaches to collision avoidance based on manoeuvres as in [Tomlin et al., 2001], this scheme would be reasonably conservative. The proposed scenario involves a number of aerial vehicles hovering above a surface upon which they can land. For example, the vehicles could be quadrotors hovering above a grass plain. The two modes involved for vehicles are: **grounded** and **airborne**. It is assumed that any airborne

vehicle can land in a bounded time, regardless of its height. That is, there is some mode transition time from **airborne** to **grounded**. It will also take an entity some time to take off i.e. become airborne from a grounded state.

The safety of such a system in terms of collision avoidance is easy to define. Two entities in **airborne** modes cannot be too close together, where the exact distance would be specified by a system designer, possibly with some redundancy added. The mode **grounded** is the only fail-safe mode of the system. Entities operating according to the Comhordú protocol should maintain collision free operation in such a system, provided all the assumptions made by Comhordú are upheld in the system. Of course, this scenario would involve three dimensional positions of entities, but Comhordú is easily generalisable to a third dimension, as discussed in Section 8.5.3.

The preceding examples are arguably somewhat contrived. Nor is there any relationship between these examples and real-world applications from the literature. Still, they are instructive to aid in the understanding of the workings of the Comhordú protocol. What is important is the common theme running through all of these examples, characterising the type of systems to which Comhordú is applicable. That is, there is always at least one fail-safe mode, one non-fail-safe mode and one "incompatibility" i.e. a situation where two entities in non-fail-safe modes can cause a safety violation if too close together.

Furthermore, a key element of these systems is that it must be the case that only *one* entity need change its behaviour to preempt an incompatibility relative to itself. Otherwise the notion of responsibility, which is a cornerstone of Comhordú is no longer enough to ensure safety and it is hard to see how safety could be achieved without consensus i.e. agreement among entities. For example, "classical" collision avoidance does not seem achievable with this protocol. That is, an entity cannot control if another entity is going to crash into it. Note the term "classical" is used here to distinguish from the earlier rather restrictive example of collision avoidance among aerial vehicles which are allowed to land. Landing is a specialised avoidance tactic, only possible when a vehicle can remove itself from the vicinity of other entities altogether. Generally speaking though, this form of avoidance is not always be possible e.g. with land-based vehicles or with aerial vehicles travelling over water.

Let us now review a small selection of literature related to application domains for Comhordú. In particular, the areas of robotics and unmanned aerial vehicles (UAVs) are examined. UAVs can be viewed as a sub-discipline of robotics. In light of the above toy examples, UAVs seem like particularly suitable candidates for applications of Comhordú. Detailed below are some examples of research in robotics and UAVs.

- In [Mostofi, 2013], techniques are devised for robot systems towards the goal of see-through cooperative object mapping. That is, robot teams are given the ability to map occluded objects. The techniques are simulated and tested in a controlled environment.

- In [Klein et al., 2011] a swarm of fish communicate via a tether i.e. a cable connecting them together. Coordination is achieved via communication over this tether. The goal is to track and follow a target.

- "Robots for Environmental Monitoring" are surveyed in [Dunbabin and Marques, 2012] with a particular focus on "marine, terrestrial and airborne robotic systems". Applications include "tsunamis, hurricanes, floods, large forest fires, volcanic eruptions, oil spills and nuclear meltdowns".

- In [Wei et al., 2013], a framework is proposed for mission assignment and scheduling in systems of UAVs. A key idea is that there are two types of agent in a system: swarm control agents and UAVs. The swarm control agents provide centralised organisation to the system while the UAVs follow. The applications of UAVs mentioned in this paper are "enemy territory reconnaissance, remote area surveillance and hazardous environment monitoring".

- A system combining UAVs and unmanned ground vehicles (UGVs) is evaluated in [Weaver et al., 2013] in terms of its efficacy in achieving certain goals e.g. search and rescue. The vehicles run the robot operating system (ROS). The system is practically evaluated in a controlled environment.

- Quadrotors are airborne vehicles with four rotors, affording them greater agility than a single rotor machine i.e. a helicopter. Miniaturisation of these devices is proposed in [Kushleyev et al., 2013] towards the goal of improved agility. The new design of quadrotor put forth is 75g in weight, which is significantly smaller than previous models. The hardware and software of a single quadrotor are first presented in the paper, followed by an explanation of the coordination architecture for a swarm. Experiments are then performed to evaluate the quadrotor team e.g. the quadrotors entering specific formations such as a spiral or the team navigating a path through a narrow window slot.

- Another paper [Alvissalim et al., 2012] investigates the applicability of quadrotor swarms toward the patching of telecommunications links that have gone down in a disaster zone. The vehicles self-deploy, acting as telecommunications links where coverage is no longer available due to some natural disaster. The use of aerial vehicles in these scenarios can surmount the difficulties faced in reaching such zones by land e.g. due to restrictions caused by debris. Retaining coverage in such zones is critical so that information can be passed on to search and rescue teams.

It is imaginable that many of these robotics/UAV examples could be extended to use Comhordú. For example, the final paper discussed [Alvissalim et al., 2012] involving quadrotor swarms for use in a disaster coverage area could potentially be augmented with the Comhordú to achieve collisions avoidance. This could be done along the same lines as the "toy" example involving collision avoidance of aerial vehicles i.e. where landing constitutes transition to a fail-safe mode. Note that what is proposed here is that Comhordú enhance rather than replace existing schemes. The Comhordú protocol is intended solely for the achievement of safety. For the pursuit of goals beyond safety a higher-level scheme/protocol is needed.

This composition of Comhordú with a higher level protocol warrants investigation in its own right. By this what is meant is examining how Comhordú behaves "in parallel" with another protocol, in some sense. On its own, Comhordú does not govern entity behaviour beyond randomly switching modes and ensuring that quite an abstract notion of safety is maintained. A higher level protocol could take into account concerns such as efficiency, fairness, task planning etc. For example, it might replace the "random" mode switching behaviour or even the random movement of entities characterised by Comhordú with more specific behaviours e.g. following a pre-determined trajectory. The exact details of how such a protocol could be composed with Comhordú, and the nature of the guarantees such a protocol would provide are left for future investigations.

8.5.3 Other Avenues for Future Work

There exist a number of other ideas for future work and extensions of this work. These are briefly outlined in the following.

- More detail could be added to the informal proof. That is, the results comprising the proof "on paper" presented in Chapter 6 could be proved or at least sketched in more detail. However, the more desirable goal is to get the machine proofs done, because the level of rigour provided by these is greater.

- More concrete coordination formal models could be attempted e.g. a system of two cars and an emergency vehicle on a road. There already exists work along these lines: CwoRIS is a model of coordination for a junction scenario that is formalised in [Asplund et al., 2012] and discussed in Section 2.2.

- Simulation constitutes an alternative direction to proof for exploring a formal model. For simulation, concrete parameters of the system would need to be specified e.g. the number of entities or the set of modes. Simulation could be done using a tool such as netlogo [Wilensky, 1999], which is "particularly well suited for modeling complex systems evolving over time" [Tisue and Wilensky, 2004]. It is possible that such simulations might uncover unforeseen behaviours in the model. Alternatively, there is the option of deploying Comhordú in a real-world scenario as discussed in Section 8.5.2, but it would be desirable to do some simulation first in order to "iron out" any potential difficulties that may arise rather

than be met with these difficulties after much time and effort has been invested into deploying a physical system.

- Robustness analysis such as that performed by the ShrinkTech [Sankur, 2013] tool is potential future work for Comhordú. Robustness analysis involves testing whether a formal model is sensitive to minor perturbations in its parameters. If this is the case for Comhordú, then it would need to be modified if it is hoped to be deployed in a real-world setting. It is possible that such modifications might encompass factors of safety to introduce redundancy e.g. in the minimum distance of compatibility or in the wait time.

  The need for such composition/augmentation stems from the fact that Comhordú is a "reflex" protocol, so to speak. That is, high-level behaviours are beyond its scope. Comhordú entities can be thought of as reflex or reactive agents which are defined as follows: "Reflex or reactive agents simply act. Therefore when cooperation occurs it does so without reflection upon possible actions. There is no prediction or predictive planning" [Doran et al., 1997]. While reflex behaviour is useful for maintaining safety, e.g. an animal flinching away from a hot material touching its skin, higher level behaviours are generally needed to achieve goals.

- It is likely that much of the methodology developed here is reusable. For example, the language would allow another protocol to be specified in the software fragment and explored in terms of its own properties e.g. a different version of safety. Alternatively, it would be possible to use the software language in its own right, i.e. separate from the other tiers of the language, for more general-purpose software specifications independent of the network level semantics. Such a venture could in particular exploit many of the machine-proved properties that already exist for this language. The only caveat to doing so would be that the process definitions would need to be redefined to the specifications of the user. That is, the current set of definitions constitute the specific Comhordú protocol, which would need to be replaced with a different, application-specific set of definitions. Nevertheless this should not be a problem because most of the proofs of language properties do not explicitly rely on this exact set of definitions.

- In a similar vein to the previous point, the Comhordú protocol could be modified but in some sense maintained in terms of its behaviour. The hope would be that a "simpler" version of the protocol in some sense exists exhibiting the same behaviour as Comhordú. For this to be achieved, the very notion of "same behaviour" would need to be formally characterised, e.g. by means of an equivalence relation such as a bisimulation such as those given in [Hennessy and Rathke, 1995]. The equivalence relation would need to be chosen such that replacing a process with an equivalent process in some sense preserves all the properties proved about the original- hence the new equivalent process would inherit safety from the original. The converse of this is that an equivalent formulation of Comhordú could be conceived for which the safety result is more simply proved. Again if properties of equivalent process are preserved by the equivalence in question, then safety for the simplified protocol would imply safety of the original. It is even imaginable that an algorithm could be formulated to automatically convert a process to its "simplest" equivalent form in some sense. Perhaps such algorithms already exist; this is a topic for further research.

- More properties could be proved about the current model. At the moment, safety is the only property that has been proved, but it would also be desirable to prove properties such as freedom from deadlock or timelock.

- Tidying up the code is a possibility for improvement in this model. It seems that a lot of the code comprising the Coq development could be more cleverly constructed e.g. using more abstraction. For example, structures such as history relations or process algebras could be defined abstractly and instantiated for the particular relations and languages appearing in the model. Abstraction allows for reusability. It also makes the presentation more succinct. One drawback of abstraction though, and the reason why code

can never be perfectly abstract the first time it is written, is that there is overhead in developing abstractions that might not pay off with the advantage of reusability. Also, the added "wrappers" that result from abstraction necessitate a good deal of "unwrapping" in theorems whereas less abstract structures are generally "flatter" and so easier to work with.

- Refining the rules of mobility of entities could enrich the model. For example, entities could be enhanced with a velocity/acceleration as opposed to just a position, thus making the dynamics of movement more realistic. Doing so would in fact restrict the set of possible behaviours. At the moment, the allowable behaviours form a superset of those that could possibly occur in reality e.g. with the current rules an entity can jump from one point to another at maximum speed and then back again immediately in the same space of time. For this, the entity would need to have infinite acceleration, which is obviously unrealistic.

  Constraining the behaviour of entities to more realistic behaviours has the potential of allowing for more progress. For example, entities could include in messages not only their position but also their velocity and even acceleration. This would provide receiving entities with more information than simply the position of the sender, allowing them to deduce with greater accuracy whether or not the threat of incompatibility is posed. For instance, a sender and receiver may be moving away from each other, meaning that even if they are in the "possibly incompatible" zones of each other, this may soon not be the case.

  Another possibility along these lines would be to constrain the movement of entities based on their modes. This would bring the formal model closer to the original. The idea is not to add specific constraints e.g. $speed < 10$ which would detract from the generality of the model. Rather, it is proposed that general, uninterpreted relations between modes and movement might be assumed to exist. It might be further assumed that these relations entail a maximum speed for each mode, as is given in the original model.

- One question that may be raised is, can some non-fail-safe state be reached? That is, can a state be reached in which there is at least one entity that is not in a fail-safe mode? If this is not the case, the system is trivial- entities simply remain in fail-safe states always and so safety is by definition maintained. Intuitively, it is believed that it is the case that for at least *some* instance of the Comhordú model a non-fail-safe state can be reached. To demonstrate why this is probably the case, consider a system with just two entities for simplicity. Now imagine one entity simply remains in a fail-safe state and the other entity begins a transition to a non-fail-safe state. Also, let us assume that the coverage of the system remains sufficient for the entire duration of this scenario. Then clearly the entity transitioning to a non-fail-safe state will succeed in doing so, because neither a coverage degradation nor a possibly incompatible message from another entity can thwart its progress.

  Of course, it would be favourable to prove formally that it is possible for some system of entities to reach a non-fail-safe state. Ideally the proof would be done in Coq. Such a proof would require the construction of an initial network and a trace leading this network to a state in which one of its constituent entities is non-fail-safe. The preceding example involving just two entities seems like the simplest candidate for such a demonstration.

- The multi-tiered language used to model Comhordú could be explored in its own right. More "sanity" properties beyond those already in existence could be proved. Equivalence relations and properties relating to these could be developed. Also, it is plausible that there is room for improvement in the language. For example, positions could be defined more abstractly rather than as two dimensional points in space.

- Probabilistic modelling of Comhordú would introduce more realistic behaviours into the model. That is, in reality, nothing is absolutely certain, and so the semantics as they are currently encoded are not entirely faithful to the real-world. The details of how probabilistic elements could be incorporated into

the model are not expounded here. Still, it is believed that the idea of adding probabilities to the model does warrant further exploration.

- It would be possible to explore a more complicated protocol based on a contract with transfer with feedback. The software language could be used to specify such a protocol. Recall that a contract with feedback is characterised by entities waiting to hear back from their neighbours before acting. Hence with this protocol, there would be two types of messages, e.g. *send* and *acknowledge*.

- An alternative direction would be to explore application specific behavioural models. For example, a formal model for traffic-based applications could be developed incorporating a characterisation of road infrastructure. It is feasible that such a model could facilitate a richer semantics. For example, rather than entities being allowed to move freely in space, they might be constrained to moving in straight lines along roads. This is in fact a standard approach in contemporary formal methods applications to motorway systems e.g. [Loos et al., 2011].

Bibliography

[Aceto, 2007] Aceto, L. (2007). *Reactive systems: modelling, specification and verification*. Cambridge Univ Pr.

[Althoff et al., 2010] Althoff, M., Althoff, D., Wollherr, D., and Buss, M. (2010). Safety verification of autonomous vehicles for coordinated evasive maneuvers. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 1078–1083.

[Alur et al., 1994] Alur, R., Courcoubetis, C., Henzinger, T., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. (1994). The algorithmic analysis of hybrid systems. In Cohen, G. and Quadrat, J.-P., editors, *11th International Conference on Analysis and Optimization of Systems Discrete Event Systems*, volume 199 of *Lecture Notes in Control and Information Sciences*, pages 329–351. Springer Berlin Heidelberg.

[Alur and Dill, 1994] Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235.

[Alur et al., 2000] Alur, R., Grosu, R., Hur, Y., Kumar, V., and Lee, I. (2000). Modular specification of hybrid systems in charon. In Lynch, N. and Krogh, B., editors, *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 6–19. Springer Berlin Heidelberg.

[Alur and Henzinger, 1992] Alur, R. and Henzinger, T. (1992). Logics and models of real time: A survey. In de Bakker, J., Huizing, C., de Roever, W., and Rozenberg, G., editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer Berlin Heidelberg.

[Alur and Henzinger, 1994] Alur, R. and Henzinger, T. (1994). A really temporal logic. *Journal of the ACM (JACM)*, 41(1):181–204.

[Alvissalim et al., 2012] Alvissalim, M., Zaman, B., Hafizh, Z., Ma'sum, M., Jati, G., Jatmiko, W., and Mursanto, P. (2012). Swarm quadrotor robots for telecommunication network coverage area expansion in disaster area. In *SICE Annual Conference (SICE), 2012 Proceedings of*, pages 2256–2261.

[André et al., 2013] André, É., Liu, Y., Sun, J., Dong, J. S., and Lin, S.-W. (2013). Psyhcos: parameter synthesis for hierarchical concurrent real-time systems. In *Computer Aided Verification*, pages 984–989. Springer.

[Arnaout and Bowling, 2011] Arnaout, G. and Bowling, S. (2011). Towards reducing traffic congestion using cooperative adaptive cruise control on a freeway with a ramp. *Journal of Industrial Engineering and Management*, 4(4):699–717.

[Asplund et al., 2012] Asplund, M., Manzoor, A., Bouroche, M., Clarke, S., and Cahill, V. (2012). A formal approach to autonomous vehicle coordination. In Giannakopoulou, D. and Mry, D., editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 52–67. Springer Berlin Heidelberg.

[Asplund and Nadjm-Tehrani, 2012] Asplund, M. and Nadjm-Tehrani, S. (2012). Analysing delay-tolerant networks with correlated mobility. In Li, X.-Y., Papavassiliou, S., and Ruehrup, S., editors, *Ad-hoc, Mobile,*

and Wireless Networks, volume 7363 of *Lecture Notes in Computer Science*, pages 83–97. Springer Berlin Heidelberg.

[Asplund, Mikael Nadjm-Tehrani, 2012] Asplund, Mikael Nadjm-Tehrani, S. (2012). Worst-case Latency of Broadcast in Intermittently Connected Networks. *International Journal of Ad Hoc and Ubiquitous Computing*, 11:125 – 138.

[Barnat et al., 2010] Barnat, J., Brim, L., Ceska, M., and Rockai, P. (2010). Divine: Parallel distributed model checker. In *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, pages 4–7.

[Barrett et al., 2009] Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2009). Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885.

[Behrmann et al., 2004] Behrmann, G., David, A., and Larsen, K. (2004). A tutorial on uppaal. In Bernardo, M. and Corradini, F., editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg.

[Bergstra et al., 2001a] Bergstra, J., Ponse, A., and Smolka, S. (2001a). *Handbook of process algebra*, chapter 7, pages 427–478. Elsevier Science.

[Bergstra et al., 2001b] Bergstra, J., Ponse, A., and Smolka, S. (2001b). *Handbook of process algebra*. Elsevier Science.

[Bertolino et al., 2012] Bertolino, A., De Angelis, G., Di Giandomenico, F., Marchetti, E., Sabetta, A., and Spoletini, P. (2012). Verification and analysis of autonomic systems for networked enterprises. In Anastasi, G., Bellini, E., Di Nitto, E., Ghezzi, C., Tanca, L., and Zimeo, E., editors, *Methodologies and Technologies for Networked Enterprises*, volume 7200 of *Lecture Notes in Computer Science*, pages 143–169. Springer Berlin Heidelberg.

[Bertot, 1998] Bertot, Y. (1998). A certified Compiler for an Imperative Language. Technical Report RR-3488, INRIA.

[Bhandal et al., 2011a] Bhandal, C., Bouroche, M., and Hughes, A. (2011a). An abstract model of a coordination protocol using the uppaal model checker. In *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on*, pages 306–311. IEEE.

[Bhandal et al., 2011b] Bhandal, C., Bouroche, M., and Hughes, A. (2011b). A process algebraic description of a temporal wireless network protocol. *The Fourth International Workshop on Formal Methods for Interactive Systems (FMIS 2011)*.

[Bouroche, 2007] Bouroche, M. (2007). *Real-time coordination of mobile autonomous entities*. PhD thesis, School of Computer Science and Statistics, Trinity College Dublin.

[Bouroche and Cahill, 2008] Bouroche, M. and Cahill, V. (2008). We don't need to agree to coordinate. In *Workshop on Dependable Network Computing and Mobile Systems (DNCMS08)*, volume 1, pages 47–51.

[Bouroche et al., 2006] Bouroche, M., Hughes, B., and Cahill, V. (2006). Building reliable mobile applications with space-elastic adaptation. *Proceedings - WoWMoM 2006: 2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2006(Mdc):627–632.

[Brucker and Wolff, 2013] Brucker, A. and Wolff, B. (2013). On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721.

[Brunel and Cazin, 2012] Brunel, J. and Cazin, J. (2012). Formal verification of a safety argumentation and application to a complex uav system. In Ortmeier, F. and Daniel, P., editors, *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 307–318. Springer Berlin Heidelberg.

[Bruttomesso et al., 2008] Bruttomesso, R., Cimatti, A., Franzn, A., Griggio, A., and Sebastiani, R. (2008). The mathsat4 smt solver. In Gupta, A. and Malik, S., editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 299–303. Springer Berlin Heidelberg.

[Cahill et al., 2008] Cahill, V., Senart, A., Schmidt, D. C., Weber, S., Harrington, A., and Hughes, B. (2008). The managed motorway: Real-time vehicle scheduling: A research agenda. In *Proceedings of the 9th Workshop on Mobile Computing Systems and Applications*, HotMobile '08, pages 43–48, New York, NY, USA. ACM.

[Calude, 2009] Calude, C. (2009). A Dialogue with Professor Joseph Sifakis about Concurrent Systems Specification and Verification.

[Carle et al., 2013] Carle, P., Choppy, C., Kervarc, R., and Piel, A. (2013). Safety of Unmanned Aircraft Systems Facing Multiple Breakdowns. In Choppy, C. and Sun, J., editors, *1st French Singaporean Workshop on Formal Methods and Applications (FSFMA 2013)*, volume 31 of *OpenAccess Series in Informatics (OASIcs)*, pages 86–91, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Cerone et al., 2013] Cerone, A., Hennessy, M., and Merro, M. (2013). Modelling mac-layer communications in wireless systems. In De Nicola, R. and Julien, C., editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg.

[Chaochen et al., 1991] Chaochen, Z., Hoare, C., and Ravn, A. P. (1991). A calculus of durations. *Information Processing Letters*, 40(5):269 – 276.

[Chen et al., 2012] Chen, X., Abrahám, E., and Sankaranarayanan, S. (2012). Taylor model flowpipe construction for non-linear hybrid systems. In *RTSS*, pages 183–192. Citeseer.

[Chen et al., 2013] Chen, X., brahm, E., and Sankaranarayanan, S. (2013). Flow*: An analyzer for non-linear hybrid systems. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 258–263. Springer Berlin Heidelberg.

[Ciaffaglione and Honsell, 2003] Ciaffaglione, A. and Honsell, F. (2003). Certified reasoning on real numbers and objects in co-inductive type theory. *Dottorato di ricerca in informatica, Dipartimento di Matematica e Informatica, Universita di Udine, Udine, Italy*.

[Cimatti et al., 2002] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). Nusmv 2: An opensource tool for symbolic model checking. In Brinksma, E. and Larsen, K., editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer Berlin Heidelberg.

[Cimatti et al., 2012] Cimatti, A., Roveri, M., Susi, A., and Tonetta, S. (2012). Validation of requirements for hybrid systems. *ACM Transactions on Software Engineering and Methodology*, 21(4):1–34.

[Cimatti et al., 2009] Cimatti, A., Roveri, M., and Tonetta, S. (2009). Requirements validation for hybrid systems. In Bouajjani, A. and Maler, O., editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 188–203. Springer Berlin Heidelberg.

[Clarke, 2008] Clarke, E. (2008). The birth of model checking. In Grumberg, O. and Veith, H., editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg.

[Clarke and Emerson, 1982] Clarke, E. and Emerson, E. (1982). Design and synthesis of synchronization skeletons using branching time temporal logic. In Kozen, D., editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg.

[Clarke and Wing, 1996] Clarke, E. M. and Wing, J. M. (1996). Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643.

[Cobano et al., 2013] Cobano, J., Alejo, D., Vera, S., Heredia, G., and Ollero, A. (2013). Multiple gliding uav coordination for static soaring in real time applications. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 790–795.

[Coupet-Grimal, 2003] Coupet-Grimal, S. (2003). An axiomatization of linear temporal logic in the calculus of inductive constructions. *Journal of Logic and Computation*, 13(6):801–813.

[da Silva, 2012] da Silva, P. (2012). *Verification of behaviourist multi-agent systems by means of formally guided simulations*. PhD thesis, Instituto de Matemática e Estatística.

[Damm et al., 2007] Damm, W., Mikschl, A., Oehlerking, J., Olderog, E.-R., Pang, J., Platzer, A., Segelken, M., and Wirtz, B. (2007). Automating verification of cooperation, control, and design in traffic applications. In Jones, C., Liu, Z., and Woodcock, J., editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 115–169. Springer Berlin Heidelberg.

[Davies and Schneider, 1995] Davies, J. and Schneider, S. (1995). A brief history of timed {CSP}. *Theoretical Computer Science*, 138(2):243 – 271. Meeting on the mathematical foundation of programing semantics.

[De Moura and Bjørner, 2009] De Moura, L. and Bjørner, N. (2009). Satisfiability modulo theories: An appetizer. In *Formal Methods: Foundations and Applications*, pages 23–36. Springer.

[de Moura and Bjrner, 2008] de Moura, L. and Bjrner, N. (2008). Z3: An efficient smt solver. In Ramakrishnan, C. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg.

[de Moura et al., 2004] de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., and Tiwari, A. (2004). Sal 2. In *Computer aided verification*, pages 496–500. Springer.

[De Wulf et al., 2004] De Wulf, M., Doyen, L., and Raskin, J.-F. (2004). Almost asap semantics: From timed models to timed implementations. In *Hybrid Systems: Computation and Control*, pages 296–310. Springer.

[Delzanno et al., 2010] Delzanno, G., Sangnier, A., and Zavattaro, G. (2010). Parameterized verification of ad hoc networks. In Gastin, P. and Laroussinie, F., editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 313–327. Springer Berlin Heidelberg.

[Detlefs et al., 2005] Detlefs, D., Nelson, G., and Saxe, J. B. (2005). Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473.

[Donz, 2010] Donz, A. (2010). Breach, a toolbox for verification and parameter synthesis of hybrid systems. In Touili, T., Cook, B., and Jackson, P., editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 167–170. Springer Berlin Heidelberg.

[Doran et al., 1997] Doran, J. E., Franklin, S., Jennings, N. R., and Norman, T. J. (1997). On cooperation in multi-agent systems. *The Knowledge Engineering Review*, 12(03):309–314.

[Dunbabin and Marques, 2012] Dunbabin, M. and Marques, L. (2012). Robots for environmental monitoring: Significant advancements and applications. *Robotics Automation Magazine, IEEE*, 19(1):24–39.

[Einstein, 1921] Einstein, A. (1921). Geometry and experience. *On a Heuristic Point of View about the Creation and Conversion of Light 1 On the Electrodynamics of Moving Bodies 10 The Development of Our Views on the Composition and Essence of Radiation 11 The Field Equations of Gravitation 19 The Foundation of the Generalised Theory of Relativity 22*, page 82.

[Eker et al., 2004] Eker, S., Meseguer, J., and Sridharanarayanan, A. (2004). The maude ltl model checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187.

[Emerson, 1990] Emerson, E. A. (1990). Handbook of theoretical computer science (vol. b). In van Leeuwen, J., editor, *Handbook of theoretical computer science*, volume B., chapter Temporal and Modal Logic, pages 995–1072. MIT Press, Cambridge, MA, USA.

[Ene and Muntean, 2001] Ene, C. and Muntean, T. (2001). A broadcast-based calculus for communicating systems. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 1516–1525.

[Felty, 2002] Felty, A. (2002). Two-level meta-reasoning in coq. In Carreo, V., Muoz, C., and Tahar, S., editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 198–213. Springer Berlin Heidelberg.

[Fillitre et al., 2001] Fillitre, J.-C., Owre, S., Rue*B, H., and Shankar, N. (2001). Ics: Integrated canonizer and solver? In Berry, G., Comon, H., and Finkel, A., editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer Berlin Heidelberg.

[Florian et al., 2012] Florian, M., Gamble, G., and Holzmann, G. (2012). Logic Model Checking of Time-Periodic Real-Time Systems. In *Infotech@ Aerospace 2012*, pages 1–8. Aerospace Research Central.

[Fränzle and Herde, 2006] Fränzle, M. and Herde, C. (2006). HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198.

[Frehse, 2005] Frehse, G. (2005). Phaver: Algorithmic verification of hybrid systems past hytech. In Morari, M. and Thiele, L., editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer Berlin Heidelberg.

[Furia et al., 2010] Furia, C. a., Mandrioli, D., Morzenti, A., and Rossi, M. (2010). Modeling time in computing. *ACM Computing Surveys*, 42(2):1–59.

[Godskesen, 2007] Godskesen, J. (2007). A calculus for mobile ad hoc networks. In Murphy, A. and Vitek, J., editors, *Coordination Models and Languages*, volume 4467 of *Lecture Notes in Computer Science*, pages 132–150. Springer Berlin Heidelberg.

[Hennessy and Rathke, 1995] Hennessy, M. and Rathke, J. (1995). Bisimulations for a calculus of broadcasting systems. *CONCUR'95: Concurrency Theory*, 200:225–260.

[Hennessy and Regan, 1995] Hennessy, M. and Regan, T. (1995). A process algebra for timed systems. *Information and Computation*, 117(2):221 – 239.

[Henzinger, 2000] Henzinger, T. (2000). The theory of hybrid automata. In Inan, M. and Kurshan, R., editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series*, pages 265–292. Springer Berlin Heidelberg.

[Henzinger et al., 2000] Henzinger, T., Horowitz, B., Majumdar, R., and Wong-Toi, H. (2000). Beyond hytech: Hybrid systems analysis using interval numerical methods. In Lynch, N. and Krogh, B., editors, *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 130–144. Springer Berlin Heidelberg.

[Henzinger et al., 1991] Henzinger, T., Manna, Z., and Pnueli, A. (1991). Temporal proof methodologies for real-time systems. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 353–366, New York, NY, USA. ACM.

[Henzinger et al., 1997] Henzinger, T. A., Ho, P.-H., and Wong-Toi, H. (1997). Hytech: A model checker for hybrid systems. In *Computer aided verification*, pages 460–463. Springer.

[Henzinger et al., 2001] Henzinger, T. A., Preussig, J., and Wong-Toi, H. (2001). Some lessons from the hytech experience. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 3, pages 2887–2892. IEEE.

[Hilaire et al., 2010] Hilaire, V., Lauri, F., Gruer, P., Koukam, A., and Rodriguez, S. (2010). Formal specification of an immune based agent architecture. *Engineering Applications of Artificial Intelligence*, 23(4):505–513.

[Holzmann, 1997] Holzmann, G. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295.

[Honsell et al., 2001] Honsell, F., Miculan, M., and Scagnetto, I. (2001). π-calculus in (Co) inductive-type theory. *Theoretical computer science*, 253:239–285.

[How et al., 2004] How, J., King, E., and Kuwata, Y. (2004). Flight demonstrations of cooperative control for UAV teams. *AIAA 3rd unmanned unlimited technical conference, workshop and exhibit*.

[Huet et al., 2007] Huet, G., Kahn, G., and Paulin-mohring, C. (2007). The Coq Proof Assistant A Tutorial. *Rapport Technique*.

[Hughes, 2006] Hughes, B. (2006). *Hard real-time communication for Mobile Ad hoc Networks*. PhD thesis, University of Dublin, Trinity College.

[Hunt, 2012] Hunt, J. J. (2012). The practical application of formal methods: where is the benefit for industry? In *Formal Verification of Object-Oriented Software*, pages 22–32. Springer.

[Jackson, 2006] Jackson, D. (2006). *Software Abstractions: logic, language and analysis*. MIT Press.

[Jiang et al., 2011] Jiang, J., Lai, Y., and Deng, F. (2011). Mobile robot coordination and navigation with directional antennas in positionless wireless sensor networks. *International journal of ad hoc and ubiquitous computing*, 7:272 – 280.

[Julien and Roman, 2004] Julien, C. and Roman, G.-C. (2004). Active coordination in ad hoc networks. In De Nicola, R., Ferrari, G.-L., and Meredith, G., editors, *Coordination Models and Languages*, volume 2949 of *Lecture Notes in Computer Science*, pages 199–215. Springer Berlin Heidelberg.

[Kesten et al., 2000] Kesten, Y., Manna, Z., and Pnueli, A. (2000). Verification of clocked and hybrid systems. *Acta Informatica*, 36(11):837–912.

[Khan et al., 2014] Khan, A. H., Khan, Z. H., and Weiguo, Z. (2014). Model-based verification and validation of safety-critical embedded real-time systems: Formation and tools. In *Embedded and Real Time System Development: A Software Engineering Perspective*, pages 153–183. Springer.

[Kleijnen, 1995] Kleijnen, J. P. (1995). Verification and validation of simulation models. *European Journal of Operational Research*, 82(1):145–162.

[Klein et al., 2011] Klein, D. J., Gupta, V., and Morgansen, K. A. (2011). Coordinated control of robotic fish using an underwater wireless network. In *Wireless Networking Based Control*, pages 323–339. Springer.

[Kushleyev et al., 2013] Kushleyev, A., Mellinger, D., Powers, C., and Kumar, V. (2013). Towards a swarm of agile micro quadrotors. *Autonomous Robots*, 35(4):287–300.

[Lamport, 1994] Lamport, L. (1994). The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923.

[Larsen et al., 1997] Larsen, K., Pettersson, P., and Yi, W. (1997). UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152.

[Lattner et al., 2005] Lattner, A., Gehrke, J., Timm, I., and Herzog, O. (2005). A knowledge-based approach to behavior decision in intelligent vehicles. In *Intelligent Vehicles Symposium, 2005. Proceedings. IEEE*, pages 466–471.

[Lomuscio and Raimondi, 2006] Lomuscio, A. and Raimondi, F. (2006). mcmas: A model checker for multi-agent systems. In Hermanns, H. and Palsberg, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 450–454. Springer Berlin Heidelberg.

[Loos et al., 2011] Loos, S., Platzer, A., and Nistor, L. (2011). Adaptive cruise control: Hybrid, distributed, and now formally verified. In Butler, M. and Schulte, W., editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 42–56. Springer Berlin Heidelberg.

[lveczky and Meseguer, 2004] lveczky, P. and Meseguer, J. (2004). Specification and analysis of real-time systems using real-time maude. In Wermelinger, M. and Margaria-Steffen, T., editors, *Fundamental Approaches to Software Engineering*, volume 2984 of *Lecture Notes in Computer Science*, pages 354–358. Springer Berlin Heidelberg.

[Marinescu et al., 2012] Marinescu, D., Curn, J., Bouroche, M., and Cahill, V. (2012). On-ramp traffic merging using cooperative intelligent vehicles: A slot-based approach. In *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*, pages 900–906.

[Marinescu et al., 2010] Marinescu, D., Curn, J., Slot, M., Bouroche, M., and Cahill, V. (2010). An active approach to guaranteed arrival times based on traffic shaping. In *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on*, pages 1711–1717.

[Merro, 2009] Merro, M. (2009). An Observational Theory for Mobile Ad Hoc Networks (full version). *Information and Computation*, 207(2):194–208.

[Merro et al., 2011] Merro, M., Ballardin, F., and Sibilio, E. (2011). A timed calculus for wireless systems. *Theoretical Computer Science*, 412(47):6585 – 6611.

[Mezzetti and Sangiorgi, 2006] Mezzetti, N. and Sangiorgi, D. (2006). Towards a Calculus For Wireless Systems. *Electronic Notes in Theoretical Computer Science*, 158:331–353.

[Milner, 1982] Milner, R. (1982). *A calculus of communicating systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA.

[Mitsch et al., 2012] Mitsch, S., Loos, S. M., and Platzer, A. (2012). Towards formal verification of freeway traffic control. In *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pages 171–180. IEEE.

[Mostofi, 2013] Mostofi, Y. (2013). Cooperative Wireless-Based Obstacle/Object Mapping and See-Through Capabilities in Robotic Networks. *IEEE Transactions on Mobile Computing*, 12(5):817–829.

[Mousavi and Reniers, 2006] Mousavi, M. R. and Reniers, M. a. (2006). Prototyping SOS Meta-theory in Maude. *Electronic Notes in Theoretical Computer Science*, 156(1):135–150.

[Nanz and Hankin, 2006] Nanz, S. and Hankin, C. (2006). A framework for security analysis of mobile wireless networks. *Theoretical Computer Science*, 367(1-2):203–227.

[Nett and Schemmer, 2004] Nett, E. and Schemmer, S. (2004). An architecture to support cooperating mobile embedded systems. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 40–50, New York, NY, USA. ACM.

[Niazi and Hussain, 2011] Niazi, M. and Hussain, A. (2011). A novel agent-based simulation framework for sensing in complex adaptive environments. *Sensors Journal, IEEE*, 11(2):404–412.

[Nicollin and Sifakis, 1992] Nicollin, X. and Sifakis, J. (1992). An overview and synthesis on timed process algebras. In de Bakker, J., Huizing, C., de Roever, W., and Rozenberg, G., editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 526–548. Springer Berlin Heidelberg.

[Nicollin and Sifakis, 1994] Nicollin, X. and Sifakis, J. (1994). The algebra of timed processes, atp: Theory and application. *Information and Computation*, 114(1):131 – 178.

[O'Hara et al., 2012] O'Hara, N., Slot, M., Marinescu, D., and Čurn, J. (2012). MDDSVsim: an integrated traffic simulation platform for autonomous vehicle research. *The International Workshop on Vehicular Traffic Management for Smart Cities (VTM 2012)*.

[Ölveczky and Meseguer, 2007] Ölveczky, P. C. and Meseguer, J. (2007). Semantics and pragmatics of real-time maude. *Higher-order and symbolic computation*, 20(1-2):161–196.

[Ostroff, 1999] Ostroff, J. (1999). Composition and refinement of discrete real-time systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(1):1–48.

[Paulson, 1989] Paulson, L. C. (1989). The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397.

[Peng et al., 2012] Peng, H., Huo, M., He, Y., and Liu, Z. (2012). Multiple uavs collision avoidance trajectory coordination using distributed receding horizon optimization. In *Intelligent Control and Automation (WCICA), 2012 10th World Congress on*, pages 3999–4003.

[Plaisted, 2014] Plaisted, D. a. (2014). Automated theorem proving. *Wiley Interdisciplinary Reviews: Cognitive Science*, 5(2):115–128.

[Platzer and Quesel, 2008] Platzer, A. and Quesel, J.-D. (2008). Keymaera: A hybrid theorem prover for hybrid systems (system description). In Armando, A., Baumgartner, P., and Dowek, G., editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer Berlin Heidelberg.

[Platzer and Quesel, 2009] Platzer, A. and Quesel, J.-D. (2009). European train control system: A case study in formal verification. In Breitman, K. and Cavalcanti, A., editors, *Formal Methods and Software Engineering*, volume 5885 of *Lecture Notes in Computer Science*, pages 246–265. Springer Berlin Heidelberg.

[Pnueli and Kesten, 2002] Pnueli, A. and Kesten, Y. (2002). A deductive proof system for ctl*. In *CONCUR 2002Concurrency Theory*, pages 24–40. Springer.

[Polya, 2004] Polya, G. (2004). *How to Solve It: A New Aspect of Mathematical Method*, page 190. Princeton University Press.

[Prasad, 1995] Prasad, K. (1995). A calculus of broadcasting systems. *Science of Computer Programming*, 25(23):285 – 327. Selected Papers of ESOP'94, the 5th European Symposium on Programming.

[Prasad, 1996] Prasad, K. (1996). Broadcasting in time. In Ciancarini, P. and Hankin, C., editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 321–338. Springer Berlin Heidelberg.

[Prasad, 2006] Prasad, K. (2006). A Prospectus for Mobile Broadcasting Systems. *Electronic Notes in Theoretical Computer Science*, 162:295–300.

[Rajkumar, 2012] Rajkumar, R. (2012). A cyber-physical future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1309–1312.

[Romberg and Grimm, 2004] Romberg, J. and Grimm, C. (2004). Refinement of hybrid systems. In *Languages for system specification*, pages 315–330. Springer.

[Rosa-Velardo et al., 2006] Rosa-Velardo, F., Segura, C., and Verdejo, A. (2006). Typed Mobile Ambients in Maude. *Electronic Notes in Theoretical Computer Science*, 147(1):135–161.

[Sankur, 2013] Sankur, O. (2013). Shrinktech: A tool for the robustness analysis of timed automata. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 1006–1012. Springer Berlin Heidelberg.

[Schobbens et al., 2002] Schobbens, P.-Y., Raskin, J.-F., and Henzinger, T. (2002). Axioms for real-time logics. *Theoretical Computer Science*, 274(12):151 – 182. Ninth International Conference on Concurrency Theory 1998.

[Senart et al., 2006] Senart, A., Bouroche, M., Hughes, B., and Cahill, V. (2006). Coordination of safety-critical mobile real-time embedded systems. In *Proc. Workshop Research Directions for Security and Networking in Critical Real-Time and Embedded Systems (CRTES06)*.

[Shue and Conrad, 2013] Shue, S. and Conrad, J. M. (2013). A survey of robotic applications in wireless sensor networks. In *2013 Proceedings of IEEE Southeastcon*, pages 1–5. IEEE.

[Silva and Richeson, 2000] Silva, B. and Richeson, K. (2000). Modeling and verifying hybrid dynamic systems using CheckMate. *Proceedings of 4th International Conference on Automation of Mixed Processes*, 4:1–7.

[Singh et al., 2010] Singh, A., Ramakrishnan, C., and Smolka, S. a. (2010). A process calculus for Mobile Ad Hoc Networks. *Science of Computer Programming*, 75(6):440–469.

[Stump et al., 2002] Stump, A., Barrett, C., and Dill, D. (2002). Cvc: A cooperating validity checker. In Brinksma, E. and Larsen, K., editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer Berlin Heidelberg.

[Tesnim Abdellatif, 2010] Tesnim Abdellatif, Jacques Combaz, J. S. (2010). Model-based implementation of real-time applications. Technical Report TR-2010-14, Verimag Research Report.

[Thati et al., 2004] Thati, P., Sen, K., and Martí-Oliet, N. (2004). An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0. *Electronic Notes in Theoretical Computer Science*, 71:261–281.

[Tisue and Wilensky, 2004] Tisue, S. and Wilensky, U. (2004). Netlogo: A simple environment for modeling complexity. In *in International Conference on Complex Systems*, pages 16–21.

[Tiwari, 2012] Tiwari, A. (2012). Hybridsal relational abstracter. In Madhusudan, P. and Seshia, S., editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 725–731. Springer Berlin Heidelberg.

[Tomlin et al., 2001] Tomlin, C., Mitchell, I., and Ghosh, R. (2001). Safety verification of conflict resolution manoeuvres. *Intelligent Transportation Systems, IEEE Transactions on*, 2(2):110–120.

[Trigui et al., 2012] Trigui, S., Koubaa, A., Ben Jamaa, M., Chaari, I., and Al-Shalfan, K. (2012). Coordination in a multi-robot surveillance application using wireless sensor networks. In *Electrotechnical Conference (MELECON), 2012 16th IEEE Mediterranean*, pages 989–992.

[Verdejo and Martí-Oliet, 2006] Verdejo, A. and Martí-Oliet, N. (2006). Executable structural operational semantics in Maude. *The Journal of Logic and Algebraic Programming*, 67(1-2):226–293.

[Vries and Subbarao, 2011] Vries, E. D. and Subbarao, K. (2011). Cooperative control of swarms of unmanned aerial vehicles. *AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, 49:1–23.

[Wagatsuma et al., 2011] Wagatsuma, Y., Toda, Y., and Kubota, N. (2011). Formation behavior of multiple robots based on tele-operation. In *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*, pages 713–720.

[Walter et al., 2010] Walter, D., Tubig, H., and Lth, C. (2010). Experiences in applying formal verification in robotics. In Schoitsch, E., editor, *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 347–360. Springer Berlin Heidelberg.

[Wang et al., 2004] Wang, F. et al. (2004). Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305.

[Wang and Lu, 2012] Wang, M. and Lu, Y. (2012). A Timed Calculus for Mobile Ad Hoc Networks. *Electronic Proceedings in Theoretical Computer Science*, 105:118–134.

[Weaver et al., 2013] Weaver, J. N., Arroyo, D. A., and Schwartz, D. E. (2013). Collaborative coordination and control for an implemented heterogeneous swarm of uavs and ugvs. In *Proc. Florida Conference on Recent Advances in Robotics*.

[Webster et al., 2011] Webster, M., Fisher, M., Cameron, N., and Jump, M. (2011). Formal methods for the certification of autonomous unmanned aircraft systems. In Flammini, F., Bologna, S., and Vittorini, V., editors, *Computer Safety, Reliability, and Security*, volume 6894 of *Lecture Notes in Computer Science*, pages 228–242. Springer Berlin Heidelberg.

[Wei et al., 2013] Wei, Y., Madey, G. R., and Blake, M. B. (2013). Agent-based simulation for uav swarm mission planning and execution. In *Proceedings of the Agent-Directed Simulation Symposium*, ADSS 13, pages 2:1–2:8, San Diego, CA, USA. Society for Computer Simulation International.

[Wilensky, 1999] Wilensky, U. (1999). Netlogo. Technical report, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

[Woodcock and Davies, 1996] Woodcock, J. and Davies, J. (1996). *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Xie et al., 2011] Xie, G., Asano, A., Hei, X., Mochizuki, H., Takahashi, S., and Nakamura, H. (2011). Formal verification of communication based train control system. In *Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE), 2011 International Conference on*, pages 394–399.

[Yasuda et al., 2012] Yasuda, Y., Kubota, N., and Toda, Y. (2012). Adaptive formation behaviors of multi-robot for cooperative exploration. In *Fuzzy Systems (FUZZ-IEEE), 2012 IEEE International Conference on*, pages 1–6.

[Yi, 1991] Yi, W. (1991). Ccs + time = an interleaving model for real time systems. In Albert, J., Monien, B., and Artalejo, M., editors, *Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer Berlin Heidelberg.

[Zhang et al., 2013] Zhang, S., Asplund, M., and Cahill, V. (2013). Reliable broadcast in vehicular ad-hoc networks. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 1640–1647.

[Zhu et al., 2009] Zhu, H., Li, M., Zhu, Y., and Ni, L. (2009). Hero: Online real-time vehicle tracking. *Parallel and Distributed Systems*, 20(6):1–13.