Time-Adaptive Dynamic Software Reconfiguration for Embedded Software

Serena Fritsch

A Thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

December 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Serena Fritsch

Dated: December 01, 2010

Permission to Lend and/or Copy

Acknowledgements

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness

Charles Dickens

First and foremost, I thank my supervisor Dr. Siobhán Clarke for her guidance, and support over the years. I very much appreciate her motivation, feedback, and endless patience during my Ph.D. studies.

I also thank the other members of Lero in the Distributed Systems Group for the good collaboration, and the fruitful discussions, which helped to shape my research.

I express my gratitude to all those who read and reviewed my thesis and provided valuable suggestions, especially Ivana Dusparic, Jenny Munnelly, Melanie Bouroche, Raymond Cunningham, Andronikos Nedos, Ashley Sterritt, and Brendan Cody-Kenny. A big thank you also to Gregor Schiele for feedback on earlier stages of this research.

To all past and present members in the Distributed Systems Group, thank you for making it such a great and lively research environment to be in. A big thank you goes especially to my room mates of Lloyd 008 for lively discussions (not only) about work, and for being great collegues. I also thank the Science Foundation Ireland, for sponsoring my research.

A very special thank you to all my friends for encouraging me, each in their own way. Each one of you deserves a big thank you. My regards should go especially to my journey fellows: As'ad Salkham, Ashley Sterritt, Bartek Biskupski, Farrukh Mirza, Guoxiang Yang, Ivana Dusparic, Marcin Karpinski, Melanie Bouroche, Mithileash Mohan, and Warren Kenny. A big thank you also to Aline Senart, Ibrahim Tawfik, Liliana Abudalo, Ilona Biskupska, Malgorzata Jaksik, Maribel Salvador, Michael Moltenbrey, Peter Otte, Sabine Horn, and Steffen Maier, for their friendship and contact to the outside world. There are many more names to be mentioned here, but i hope you all know who you are.

And finally, thank you to my family for their continuous love, support, and encouragement.

Serena Fritsch

University of Dublin, Trinity College

December 2010

Summary

Reactive embedded systems, such as sensor nodes, or real-time control systems, are embedded into and interact with a physical environment for controlling and measuring purposes. When deployed in environments with changing requirements and unpredictable events, these systems need to adapt their software to maintain a desired level of functionality. As these systems are generally long lived, they need to provide means for allowing software adaptations such as introducing new, or improving old functionality. There are many static software adaptation techniques, but high reliability and durability constraints for reactive embedded systems make stopping and changing their structure or behaviour offline undesirable. Dynamic software reconfiguration is needed to allow such changes to happen while the system, or parts of the system, remain functional.

Dynamic software reconfiguration provides mechanisms that enable software structure and behaviour changes at system run-time. Such reconfiguration occurs in conjunction with normal processes in a reactive, embedded system. However, this has the potential to cause conflict between two requirements: On the one hand, even while the software of a system is being reconfigured, it needs to react to many, typically unstructured, events, including physical events, user commands, and messages from other systems. As many of these systems operate in real-time, a response to these events needs to occur within time bounds. On the other hand, a reconfiguration may be required to react to current prevailing conditions and therefore it should complete as soon as possible to maintain acceptable quality of service.

Transactional reconfiguration approaches complete the reconfiguration without consideration of the incoming events and their processing deadline. However, this might lead to a missed event deadline with implications for the system's overall timeliness. Preemptive reconfiguration approaches stop any reconfiguration activity and directly process an incoming event, which results in a timely computation of the event. However, this might delay a reconfiguration indefinitely if many incoming events occur consecutively. A reconfiguration model targeting reactive, embedded systems should fulfill two requirements: The model should meet the processing deadline of incoming events, while at the same time it should make progress towards a completion of the reconfiguration. The research question that emerges is what techniques are neccessary to realise such a reconfiguration model.

To address this question, this thesis introduces TimeAdapt, a time-adaptive reconfiguration model. Input to the model is the current software configuration, an event's processing deadline, and the remaining reconfiguration activities. Based on that input, the model determines how much of the reconfiguration can still be executed to reach a safe configuration within the given processing deadline. The core of the model is a partitioning of a reconfiguration process sequence into sub-sequences of reconfiguration tasks that are safe to be interrupted and sub-sequences that must be executed atomically to reach a valid safe intermediate system configuration. The reconfiguration model is defined for embedded software that follows the data-flow based computational model. This model was chosen as it is a well-defined mathematical model for concurrent computations and is general enough to describe distributed and local software architectures.

The contributions of this thesis are two-fold. Firstly, the time-adaptive reconfiguration model enables the timely reaction to incoming events, while at the same time progress towards a reconfiguration completion is made. Secondly, a reconfiguration system is designed and implemented, which is a real-world implementation of the time-adaptive reconfiguration model. The system executes local reconfigurations using time-bounded reconfiguration algorithms on components that follow the data-flow architecture.

The reconfiguration system is implemented for a real embedded platform, Java SunSpots, and the reconfiguration algorithms are compared to transactional and pre-emptive reconfiguration approaches for an example application scenario. The evaluations show that our model has a higher percentage of met deadlines, while it leads to a faster completion of an ongoing reconfiguration.

Publications Related to this Ph.D.

[1] Shane Brennan, Serena Fritsch, Yu Liu, Ashley Sterritt, Jorge Fox, Eamon Linehan, Cormac Driver, Rene Meier, Vinny Cahill, William Harrison, Siobhán Clarke, "A Framework for Flexible and Dependable Service-Oriented Embedded Systems", to appear in the 7th Book on Architecting Dependable Systems (ADS 7), Springer, 2010

[2] Serena Fritsch, and Siobhán Clarke, "TimeAdapt: Timely Execution of Dynamic Software Reconfigurations". Proceedings of the 5th Middleware Doctoral Symposium (Middleware 2008), pages 13-18, December 2008, Leuven, Belgium.

[3] Serena Fritsch, Aline Senart, Douglas C. Schmidt, and Siobhán Clarke. "Time-Bounded Dynamic Adaptation for Automotive System Software". Proceedings of the 30th International Conference on Software Engineering (ICSE), Experience Track on Automotive Systems, May 2008, Leipzig, Germany.

[4] Serena Fritsch, Aline Senart, Douglas C. Schmidt, and Siobhán Clarke, "Scheduling Time-Bounded Dynamic Software Adaptation". Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), at ICSE 2008, May 2008, Leipzig, Germany.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reactive, embedded systems, such as sensor nodes, or real-time control systems, are deployed in and interact with a physical environment to control and measure its variables. The software that executes on such systems is often deployed in environments with changing requirements and unpredictable events. To better cope with these changing conditions, the software needs to be adapted. However, the high reliability and durability constraints of a reactive embedded system make stopping and changing its structure or behaviour offline undesirable. Dynamic software reconfiguration provides mechanisms that enable software structure and behaviour to change at run-time, without the need to stop and restart the system. However, the reconfiguration activity is interleaved with normal processes. So, while the software of a reactive embedded system is being reconfigured, it needs to react to many, typically unstructured, events, which must be processed in real-time. The timely processing of the events has the potential to conflict with the timely completion of the reconfiguration process itself. In current approaches to the dynamic reconfiguration of embedded software, either the reconfiguration is completed without consideration of incoming events and their processing deadlines, or the completion of a reconfiguration is not guaranteed. This thesis proposes a new reconfiguration model for software deployed on embedded systems, TimeAdapt. TimeAdapt aims to meet the processing deadline of incoming events, while at the same time, making progress towards completion of a reconfiguration. This introductory chapter provides the background and motivation for this work, introduces the proposed approach to the dynamic reconfiguration of embedded software, presents the contributions of this thesis and outlines the remainder of this thesis.

1.1 Background

Embedded systems, such as sensors in a sensor network or real-time control systems, are reactive (Rutten, 2008). Unlike transformational or interactive systems, reactive systems are computer systems that continuously react to their environment at a speed determined by the environment (Halbwachs, 1993). In contrast to general-purpose software, embedded software interacts with the physical world rather than dealing with the transformation of data (Lee, 2002). According to Lee (2002), the software of reactive embedded systems has some distinctive features such as:

- **Concurrency**: A typical embedded application consists of several parallel activities that interact with each other and with the external environment (Cheong & Liu, 2005). Each of these activities must react to stimuli from a variety of sensors.
- **Event-driven Computations:** The software is event-driven, i.e., conceptually concurrent components are activated by incoming events.
- **Timeliness of Computations:** Many of these systems control critical entities and incoming events, and actuators, must be executed within a time bound.
- **Liveness Property:** In embedded systems, liveness is a critical issue as programs must not terminate or block waiting for events that may never occur (Lee, 2002). Termination is considered a failure and should be avoided.

Embedded software runs in a particular configuration, which is defined as “a particular arrangement and setup of data, software components, hardware resources, as well as their relationships and properties that allows an embedded system to operate correctly according to its architecture" (Perrson, 2009). Traditionally, this configuration was statically determined at system design-time and did not change during system runtime. However, the trend towards open embedded systems requires the dynamic change of configurations (Baresi et al., 2006). These systems are exposed to changing environments and to changes that were not envisioned at design time. The liveness requirements of reactive embedded systems makes stopping the system and changing its software configuration offline undesirable (Lee, 2002). Consequently, dynamic software reconfiguration is needed to allow such changes to occur while the software remains functional.

Dynamic software reconfiguration is defined in the literature as the evolution of a software from its current configuration to another configuration at system runtime (Aksit & Choukair, 2003). Dynamic changes of the software configuration include amongst others structural changes that impact the topology of the software by adding or removing software entities, or the modifications of interfaces or implementations of the software entities themselves. A software entity represents the most fine-grained software abstraction that is subject to structural and behavioural change (Janssens, 2006). Dynamic software reconfiguration originated in the domain of long-lived, highly available distributed systems such as banking applications, database servers, and telecommunication switches, where the software entities are processing nodes that are part of a distributed system (Kramer & Magee, 1985). However, dynamic software reconfiguration is also used on finer-grained software entities within a node, among them software procedures (Neamtiu & Hicks, 2009), object-oriented software structures (Welch & Stroud, 2000), components in component-oriented systems (Coulson et al., 2008; Seto et al., 1998) or aspects in aspect-oriented systems (Popovici et al., 2002; Truyen et al., 2008). This thesis adopts components as the building blocks of software entities in a reactive embedded system (Vandewoude, 2007). The explicit notion and reification of software composition in component-oriented systems makes them very suitable for dynamic software reconfiguration (McKinley et al., 2004). More specifically, this thesis addresses component-based embedded application software that runs on top of a single-processor embedded system (Cheong, 2007). Examples for this kind of software are sensing applications on a single sensor node. For the rest of this thesis, this kind of software is referred to as embedded software. However, the approach can also be generalised to lower-level system software, such as operating system software (Friedrich et al., 2001), or middleware software (Schmidt, 2002).

1.2 Motivation

Dynamic reconfiguration in embedded software occurs while the underlying system reacts to incoming events such as physical events, user commands, and messages from other systems (Cheong et al., 2003). For example, while the reconfiguration is executed, the underlying hardware raises an interrupt. However, these incoming events have the potential to cause conflict between two requirements. Firstly, as many of these systems operate in real-time, event processing needs to be started within the so-called processing deadline (Lee, 2002). Secondly, a reconfiguration may be required to react to current prevailing conditions and therefore it should complete as soon as possible to maintain acceptable quality of service.

Current reconfiguration models for embedded software can be classified according to their execution model into *transactional* or *preemptive* approaches, depending on how they address the two conflicting requirements of a reconfiguration.

In transactional reconfiguration approaches, a reconfiguration is either executed atomically or it is aborted. The main focus of these approaches is on the safe and timely execution of a reconfiguration. This includes the timely completion of reaching a reconfiguration safe state, where the entities are not involved in any interaction that would change their execution state, and the timely execution of the actual reconfiguration itself (Wermelinger, 1997). Some system-specific reconfiguration models, such as Port-based Objects for real-time control systems (Stewart & Khosla, 1996), execute reconfigurations within static time bounds that are either dictated by the characteristics of the underlying hardware or defined by the application developer. However, all these approaches do not react to previously unknown incoming events. As a result, the processing deadlines of these events might be missed with implications for the system's overall timeliness.

In contrast to transactional approaches, preemptive reconfiguration approaches allow the interruption of an ongoing reconfiguration (Zhao & Li, 2007b). These approaches can directly react to an event by preempting the ongoing reconfiguration and processing the event. A precondition for these approaches is that the reconfiguration can be directly pre-empted. These approaches do not support the reconfiguration of stateful software entities, as this implies reconfiguration actions that are not instantaneous. However, in the presence of many incoming events, the completion of a reconfiguration is potentially delayed indefinitely. As a result, these approaches do not meet the requirement for a timely completion of the reconfiguration process.

1.2.1 Challenges for Dynamic Software Reconfiguration in Embedded Software

Embedded software imposes additional challenges for reconfiguration models with respect to the following factors that need to be considered: Reaching quiescent state, maintaining structural dependency relationships, and scheduling reconfigurations.

- **Reaching quiescent state** A pre-condition for dynamic software reconfiguration is that after the reconfiguration has completed, the software is left in an execution state so that it can keep on functioning (Janssens, 2006). It must be ensured that affected software entities are not involved in any computation during a reconfiguration, as otherwise their execution state would change (Goudarzi & Kramer, 1996). One approach taken by many reconfiguration approaches is to freeze the entities to be reconfigured into a reconfiguration safe state, the so-called quiescent state.

Embedded software, which consists of multiple threads of execution, extends the time it takes until a quiescent state is reached. It also affects the type of approach suitable for reaching a quiescent state. For example, using an approach that reaches a quiescent state by observing system execution is not feasible as execution threads never terminate and block reconfiguration start (Wegdam, 2003).

- **Maintaining structural dependency relationships** Dependency relationships between software entities constrain the structure of the software (Almeida et al., 2001). Dynamic software reconfiguration must not break the dependencies between collaborating software entities, as otherwise the structural integrity of the software is not guaranteed.

  It is a significant challenge for an ongoing reconfiguration to maintain dependency relationships in the presence of incoming events. If the event is directly processed, the reconfiguration may be only partially executed. The partial execution of a reconfiguration might lead to a violation of dependency relationships. To illustrate this in the context of reactive embedded software, suppose an encryption software module collaborates with a decryption software module, i.e., both modules are related via a dependency relationship. A reconfiguration replaces both software modules. If the reconfiguration process is interrupted after, for example, the encryption component is replaced, the decryption module might not be able to decrypt data that is encrypted by the newly replaced encryption module. To prevent dependency breakage, the two reconfiguration actions need to be executed as one atomic action (Léger et al., 2007).

- **Scheduling reconfigurations** Dynamic software reconfiguration may negatively impact an embedded system's performance, especially when executed on single-processor platforms. On such platforms, the reconfiguration process competes with functional code for processor resources (Zhao & Li, 2007b) and either preemptive or time-sliced scheduling mechanisms are applied. These scheduling mechanisms directly determine the type of execution model a reconfiguration follows.

A preemptive scheduling mechanism assigns a high priority to functional code, such as events, and a low priority to the reconfiguration process. This leads to a preemptive reconfiguration model and to the already mentioned issues of reconfiguration starvation when the event arrival rate is high.

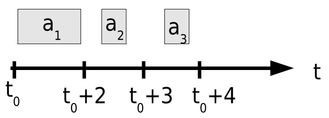
A clock-driven scheduling mechanism separates processor time into many slices. In some slices, the reconfiguration process runs, and in other slices, the functional code, and events, are executed (Zhao & Li, 2007b). A clock-driven scheduling mechanism leads to a transactional reconfiguration model, as the reconfiguration is only interrupted for statically known events. However, this mechanism is inflexible in terms of reacting to arbitrary events.

1.3 Thesis Aims and Objectives

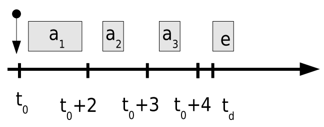
The previous challenges and the limitations of existing reconfiguration approaches motivate the research question addressed by this thesis, which is: what techniques and properties are necessary from a reconfiguration model and its underlying system model, targeting embedded software, to react to incoming events in a timely fashion, while at the same time ensuring that a reconfiguration eventually completes. In particular, this thesis addresses the challenge of maintaining structural dependency relationships in the presence of partially executed reconfigurations.

To address this question, this thesis proposes TimeAdapt, a time-adaptive reconfiguration model for embedded software. A time-adaptive reconfiguration model allows the dynamic adaptation of an ongoing reconfiguration sequence to dynamic time bounds, imposed by incoming events. TimeAdapt dynamically adapts an ongoing reconfiguration to the processing deadline of an incoming event by determining how many of the remaining reconfiguration actions can still be executed within this deadline. This implies that the overall reconfiguration itself may be executed only partially to meet a given processing deadline. Figure 1.1 illustrates the basic principle of the time-adaptive execution model by means of a reconfiguration sequence comprised of three reconfiguration actions a_1, a_2, a_3, which are to be executed sequentially. a_1 takes 2 time units to execute, whereas a_2, and a_3 take 1 time unit to execute (see Figure 1.1(a)). t_d denotes the processing deadline of an incoming event, i.e., the maximum time span, until when an event must be processed. Figure 1.1(b) illustrates the case when an event occurs before any reconfiguration action is executed, and with a processing deadline t_d, which is larger than the overall reconfiguration sequence execution duration. In this case all actions can be executed, before the event is processed. Figure 1.1(c) illustrates the case, where the event also occurs before any reconfiguration action is executed, however, its processing deadline t_d can only fit the first reconfiguration action. Hence, the remaining reconfiguration actions are executed only, after the event has been processed. Figure 1.1(d) illustrates the case, where even though the event occurs before any reconfiguration action is executed, the event processing deadline t_d is too small so that no reconfiguration action fits the deadline. Hence, the event is directly processed and the reconfiguration actions are executed after the event has been processed. Case e) illustrates the case when an event deadline is missed and is described in more detail below.

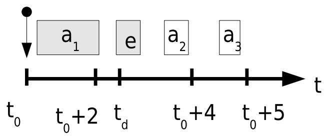
Given an event's timeliness requirement, TimeAdapt makes as much progress as possible with the reconfiguration while at the same time aiming to meet the event's processing deadline. At the core of the model is the partitioning of a reconfiguration process sequence into sub-sequences of reconfiguration actions that are safe to be interrupted and sub-sequences that must be executed atomically to reach a valid safe intermediate configuration. In all cases, the model guarantees that dependency relationships between software entities are not destroyed and that a functioning system is maintained. There is a clear tradeoff between processing all events within their given processing deadline and maintaining a valid safe configuration at all times. In TimeAdapt, the processing of events may be delayed in favour of guaranteeing an executable system that does not block -in other words, a valid safe configuration. Figure 1.1(e) illustrates the case, when



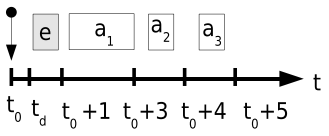
(a) Planned reconfiguration sequence



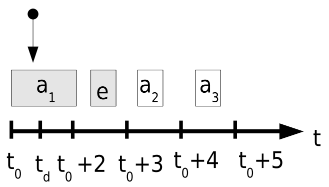
(b) Timeline when event processing deadline $t_d \geq t_0 + 4$



(c) Timeline when event processing deadline $t_0 + 2 \leq t_d < t_0 + 3$



(d) Timeline when event processing deadline $t_0 + 1 \leq t_d < t_0 + 2$



(e) Timeline when event processing deadline $0 \leq t_d < t_0 + 1$

Fig. 1.1: Execution of reconfiguration actions depending on event processing deadline t_d

an incoming event occurs during an executing reconfiguration action a_1. TimeAdapt either completes or revokes this action to reach a valid configuration. However, the event processing deadline is too small to fit either the completion or the revocation of the reconfiguration action and is missed. A missed event deadline is tolerable in software that is deployed in a soft real-time environment. For example, consider a sensing application deployed on a single sensor node platform. Sensing values received by other nodes in the network could be represented as events. These events have associated processing deadlines as stale data should be avoided. However, even if the event's processing deadline is not met, the software is functioning and can use the data for processing.

A reconfiguration in TimeAdapt is divided into two phases: a partitioning phase and a scheduling and execution phase. In the partitioning phase, all reconfiguration actions are logically mapped to sub-sequences. The dependency relationship of the software entities, contained in the reconfiguration actions, determines the size of these sub-sequences. The partitioning phase occurs at reconfiguration design time. The scheduling and execution phase takes place at reconfiguration runtime, whenever an incoming event coincides with an ongoing reconfiguration. A scheduling algorithm then decides, based on the available deadline and the estimated reconfiguration action times, how many of the remaining reconfiguration actions, now logically mapped to sub-sequences, can still be executed before an incoming event's processing deadline is reached.

A reconfiguration model is always defined for a specific model of the software, the so-called reconfiguration system model, which we abbreviate as system model. TimeAdapt is defined for embedded software that follows the Reconfigurable Dataflow System model (RDF). This theoretical system model is an extension of the conceptual data-flow process network, which is also known as actor model (Lee et al., 2003). There are two characteristics of the RDF model that motivate its use as the base model for our reconfigurable extensions. Firstly, the RDF is a mathematical approach for modelling distributed, concurrent computation (Agha, 1986). The mathematical formulation ensures that the model is unambiguous and verifiable. The RDF model is well-suited to describe the kind of software this thesis targets (Cheong, 2007). Secondly, the RDF model is generic, such that TimeAdapt is applicable to a wide range of embedded software. Over the past decade, this model has been used widely to describe highly concurrent software in various areas such as signal processing, linear and non-linear control systems, image processing systems or other stream-oriented systems (Zhao & Li, 2007b).

1.4 Contributions

The reconfiguration model described in this thesis contributes to the state of the art in the area of reconfiguration models for software deployed on reactive embedded systems in the following points:

- The dynamic approach towards reconfiguration execution allows the interruption of an ongoing reconfiguration to events that coincide with a reconfiguration. The partial execution of a reconfiguration allows the meeting of processing deadlines, given that the execution duration of the sub-sequence fits within the deadline.

- While supporting incoming events, the reconfiguration approach provides scheduling algorithms that execute as many reconfiguration actions as are possible within an incoming event's deadline, leading to an eventual completion of the reconfiguration.

- The reconfiguration model in this thesis supports stateless and stateful software entities, and its definition for an abstract, implementation-independent system model allows the application of the model for embedded software in various application domains.

A reconfiguration system has been implemented that realises TimeAdapt as part of a reconfiguration manager on a single-processor embedded platform, Java SunSpots (Sun, 2006). The evaluation of TimeAdapt on this platform validates the advantages of a time-adaptive reconfiguration model over transactional and preemptive models, with regards to meeting event deadlines and the number of completed reconfiguration actions. However, as discussed, the model is only useful for embedded software and respective systems that can deal with some events not meeting their deadlines, i.e., which have no critical deadline associated with events.

1.5 Scope

The term “reconfiguration” is often used for the change and optimisation of hardware parts such as FPGA’s (Ghiasi et al., 2005). However, the techniques and mechanisms differ strongly from the ones used in this thesis.

This thesis focusses on the actual process of reconfiguration execution. It is assumed that a reconfiguration sequence is given as input. The procedures for obtaining a reconfiguration sequence are beyond the scope of this work. It should also be noted that this work assumes that reconfigurations are valid and do not harm system consistency. Moreover, the issues of trust and security are not covered, i.e., it is assumed that reconfigurations are always for the improvement of the software and not of malicious nature.

1.6 Thesis Outline

The remainder of this thesis is organised as follows. Chapter 2 presents the state of the art in dynamic reconfiguration approaches targeting a variety of embedded software, with a particular emphasis on existing reconfiguration systems and their associated execution models. Chapter 3 presents the design of TimeAdapt and its system model. Chapter 4 presents the implementation of TimeAdapt and the mapping of the system model entities to entities of a component model implementation. Chapter 5 experimentally validates the properties of the reconfiguration model presented in Chapter 3, for software deployed on a real embedded system platform. Chapter 6 summarises and discusses future work.

1.7 Summary

This chapter outlined the goals and scope of work described in this thesis, specifically the definition of a new reconfiguration model that reacts to incoming events in a timely manner. Background information relating to embedded software and dynamic software reconfiguration that is relevant for this thesis were presented and the challenges of dynamic software reconfiguration in embedded software discussed. The problem was defined in more detail by examining the limitations of existing reconfiguration approaches. In addition, this chapter outlined the contributions and scope of this thesis.

Chapter 2

Dynamic Reconfiguration of Embedded Software

This chapter provides an overview of basic concepts found in dynamic software reconfiguration targeting embedded software in general. Based on this discussion a set of features for reconfiguration models in this domain are derived. This set of features is then used as a guide to review existing reconfiguration models targeting different kinds of embedded software.

2.1 Dynamic Software Reconfiguration of Embedded Software

This section introduces dynamic software reconfiguration targeting embedded software from three points of views: the rationale for dynamic reconfiguration; whether reconfiguration is managed by a centralised or decentralised entity; and the actual execution of reconfigurations (Hammer, 2009).

2.1.1 Dynamic Software Reconfiguration Rationale

Static configuration is commonly applied to embedded software, which is executed on platforms that allow a variety of parameters. Static configuration is defined, according to Perrson (2009), as "the possibility to easily change the configuration of a system at design time through tools." For example, the features in an automotive control system software, can be personalised according to a customer's requirements. Static configuration has been facilitated by the development of modularised software entities, so-called components, that constitute the embedded software and which allow the modularisation of functionality into different building blocks (Stewart & Khosla, 1996).

The growing complexity of embedded software extends the requirements for configuration towards reconfiguration of the software at runtime. There are two rationales for dynamic software reconfiguration. The first one is to handle software failure. A failure occurs when the software is either not functioning or only partially functioning. This includes approaches to formally specify a reconfigurable system that maintains a set of properties when software fails (Strunk & Knight, 2004), to model adaptive embedded software behaviour that can be statically verified (Trapp et al., 2007), and to provide frameworks that realise dynamic reconfiguration when there are faults in the system (Seto et al., 1998).

The second rationale is to handle changing conditions in the environment or software evolution (Soules et al., 2003), (Stewart & Arora, 1996). Reconfigurations include the update of software entities to newer versions or the change of software structure by adding and removing software entities and their connections. In contrast to recovery reconfiguration, the software is still running. As this thesis also assumes that the software is still running, while a reconfiguration is executed, adaptive and evolutionary reconfigurations are the main focus for the rest of this document.

2.1.2 Reconfiguration Management

A reconfiguration model defines the possible reconfiguration operations for software entities of a specific granularity. A prominent reconfiguration model, which has influenced the subsequent work of many other reconfiguration models, including those targeting embedded software, is Kramer and Magee's (Kramer & Magee, 1985). This model executes incremental modifications and is defined for an abstract system model, in which the system is seen as a directed graph with nodes as components and transactions between nodes denoting the connections. A centralised entity, the reconfiguration manager, has a global view on the system and allows the coordination, ordering, and management of reconfiguration actions. The centralised entity sends reconfiguration commands to the respective system entities, which then execute the actual modifications.

In domains, such as peer-to-peer networks or autonomic distributed systems that might involve the adaptation of a large number of software entities, centralised reconfiguration management might not scale well. These domains need decentralised reconfiguration approaches, such as the K-Component approach (Dowling, 2004). In decentralised reconfiguration, the distributed software entities decide, which reconfiguration actions to execute and how to execute them, so-called self-configuration. In contrast to dynamic reconfiguration, self-configuration needs additional algorithms and techniques, such as monitoring, to enable the software entities to detect when and how to execute their own changes. Self-configurable systems are a subset of reconfigurable systems, but are not explicitly dealt with in this thesis. As this thesis deals with non-distributed embedded systems, we adopt the view of a centralised entity that executes reconfigurations.

Embedded software is often logically or physically distributed onto many platforms. Reconfiguration can be categorised then into whether it affects only software entities located on a single platform, i.e., local reconfiguration, or entities distributed on multiple platforms, i.e., distributed reconfiguration (Janssens, 2006). In distributed reconfigurations, the affected platforms must agree as to which reconfiguration actions to take and ensure that these reconfigurations are executed in a safe and valid manner. For example, a multimedia conferencing application, comprised of inter-connected devices, might adapt to a different level of network bandwidth. Data must be encoded differently when the level of available network bandwidth decreases. A reconfiguration may involve the replacement of existing encoding filters on the source node, which hosts the streaming application. In this scenario, all receiver devices must replace their decoding filter components to ensure that the communication is not interrupted (Grace, 2008). The entity responsible for the coordination depends on the type of reconfiguration management applied. When using a centralised reconfiguration manager, the coordination is executed as part of this manager.

2.1.3 Reconfiguration Execution

Figure 2.1 illustrates the different actions a centralised reconfiguration manager takes when a reconfiguration request is received. In this example, the reconfiguration is of an adaptive nature and triggered when there is a change in the environment. Reconfigurations triggered for evolution purposes follow a similar sequence of steps. Reconfiguration actions such as the loading of new software entities and setting their internal state can be done without affecting the currently active configuration. Reconfiguration actions such as replacing stateful software entities affect the currently active configuration. These reconfiguration actions must ensure that the affected entities are not involved in any computation to guarantee a consistent configuration. A consistent configuration is defined as a configuration that satisfies structural integrity requirements and in which all affected entities are in mutually consistent execution states (Goudarzi & Kramer, 1996). One approach is to rely on the underlying application to deal with inconsistencies that occur during reconfiguration by providing rollback and recovery mechanisms (Vandewoude, 2007). However, this approach always executes a reconfiguration and then checks, whether this reconfiguration destroyed system consistency or other constraints (Hofmeister, 1994). As this approach does not check online, whether timing constraints are not met, it is not further considered for this thesis.

Kramer and Magee introduced the concept of quiescence as a condition for consistency and described mechanisms for how to achieve quiescence by explicitly freezing affected entities (Kramer & Magee, 1990). Once the software is in a quiescent state, reconfigurations can be safely executed, by sequentially executing the remaining reconfiguration actions.

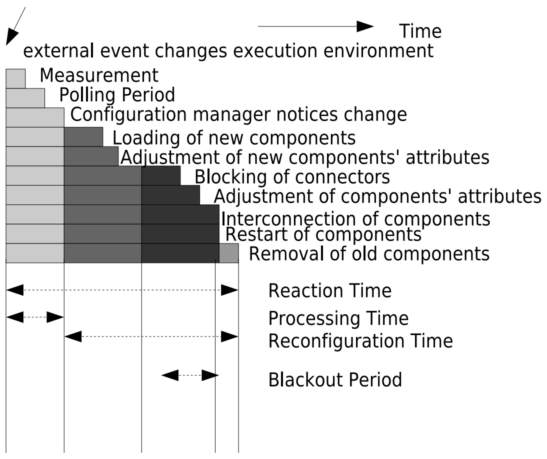

Time
external event changes execution environment
Measurement
Polling Period
Configuration manager notices change
Loading of new components
Adjustment of new components' attributes
Blocking of connectors
Adjustment of components' attributes
Interconnection of components
Restart of components
Removal of old components
Reaction Time
Processing Time
Reconfiguration Time
Blackout Period

Fig. 2.1: Timing analysis of reconfiguration process (Rasche & Polze, 2003)

While a software entity is in a quiescent state, it cannot process any computations. This is termed the blackout time of a reconfiguration (Schneider et al., 2004). As a high blackout time leads to a high system disturbance, research on dynamic reconfiguration branched into two directions (Li, 2009). One direction focusses on minimising the number of affected entities. For example, Wermelinger extended Kramer and Magee's approach to block only connections between affected entities (Wermelinger, 1997). Vandewoude relaxed Kramer and Magee's consistency approach even further by blocking only the affected software entities and not entities that could potentially initiate transactions on these entities, resulting in a very small blackout time (Vandewoude, 2007).

The second direction focusses on logically removing quiescence by applying transitional reconfiguration, which activates the new configuration before removing the old configuration (Li, 2009). The reconfiguration model then switches, at some point in time, directly between these two configurations, when there are no ongoing transactions for the old configuration. This approach needs additional mechanisms such as transaction versioning to determine when the old configuration does not process any requests and can be removed safely (Zhao & Li, 2007b).

Reconfiguration Execution Models How a reconfiguration is executed depends on the actual scheduling approach that is used in the underlying operating system. For example, a preemptive scheduling mechanism associates incoming events with a higher priority and leads to a *preemptive execution model.* The pre-condition for these models is that the reconfiguration can be directly switched with functional code, without destroying consistency. To achieve this, preemptive reconfiguration models often apply transitional reconfiguration, in which the old and the new configuration are concurrently present. A clock-driven scheduling mechanism either executes the reconfiguration or the functional code of the software, which leads to a *transactional execution model.* In this execution model a sequence of reconfiguration actions can only be interrupted by events that are known at system design time. Reconfiguration models that follow a transactional execution model are from now on denoted as transactional reconfiguration models, whereas reconfiguration models that follow a preemptive execution model are denoted as preemptive reconfiguration models.

Table 2.1 summarises the advantages and the shortcomings of both execution models with respect to the characteristics of embedded software, namely incoming events associated with processing deadlines, and the requirement of a guaranteed reconfiguration completion (see also section 1.2). A transactional execution model guarantees the completion of a reconfiguration, but it falls short in meeting the timeliness requirements of incoming events. A preemptive execution model addresses the timely response to an incoming event, but a high event arrival rate leads to a potential starvation of an ongoing reconfiguration.

Requirement/Exec. Model	Transactional	Preemptive
Guaranteed Reconf. Completion	✔	✘
Timely Event Response	✘	✔

Table 2.1: Properties of execution models

To guarantee reconfiguration completion, a preemptive execution model needs to overcome the issue of reconfiguration starvation when there is a high event arrival rate. Possible solutions include the interruption of a reconfiguration only to statically known events (Stewart et al., 1997), or by placing various priorities on incoming events, e.g., used by the preemptive scheduling mode of Zhao & Li (2007b). In the first approach, a reconfiguration preempts only events that are known at system design time. A reconfiguration can then be rejected directly, if the rate of these events exceeds a certain threshold. However, reactive embedded systems are exposed to many dynamic event sources, which emit unknown events at arbitrary times. In the second approach, incoming events are prioritised according to a given scheme. The reconfiguration model then only interrupts the ongoing reconfiguration to events of a certain priority. However, this approach assumes a specific system model in which events can be tagged and cannot be applied in more general scenarios.

A transactional reconfiguration model meets an incoming event's response deadline only if the remaining reconfiguration sequence can be completed within this deadline. Before a reconfiguration sequence is started, all events, as well as their arrival rates and deadlines must be statically known, to guarantee that their event deadlines are met (Rasche & Polze, 2005). However, this is not feasible in reactive embedded systems due to the vast occurrence of potential event sources (Regehr, 2008).

This thesis argues for a reconfiguration model that combines the advantages of transactional and preemptive execution models. The reconfiguration model should follow a preemptive execution model to allow the timely reaction to incoming events. At the same time, it should guarantee an eventual completion of a reconfiguration, even in the presence of a high event arrival rate, through an incremental execution of remaining reconfiguration actions. We define this kind of execution model *time-adaptive*, as the execution of the reconfiguration is adapted to its available time, which is constrained by incoming events. In the next section, a minimum set of features is derived that a reconfiguration model should provide in order to enable the timely reaction to incoming events and to guarantee an eventual reconfiguration completion. This set of features is then used to review to what degree existing reconfiguration models for embedded software fulfil the characteristics of a time-adaptive reconfiguration model.

2.1.4 Reconfiguration Execution Models in Hard Real-Time Systems

For completion, we briefly discuss reconfiguration execution in hard real-time systems, and why these models cannot be applied in the kind of embedded software that is considered for this thesis.

Hard real-time systems have additional timing constraints besides their functional requirements, which usually specify that a given activity must be completed within a deadline (Kopetz, 1997). In contrast to the systems we target in this thesis, deadlines in hard real-time systems are strict, and must not be missed. Multi-moded real-time applications are comprised of various operating modes, which each consist of tasks that execute a specific system functionality. For example, a fault recovery mode consists of recovery actions and re-initialisation activities for faulty tasks.

These systems realise dynamic reconfiguration via mode-changes, when there are changes in the environment or changes in the internal state of the application. A mode change removes tasks that belong to the old mode and releasing tasks that belong to the new mode. In a transitioning phase, old tasks are still active and new tasks are scheduled into the system. To guarantee that all tasks reach their deadlines, mode-change protocols are used that differ as to when to delete old tasks and when to schedule new tasks for execution (Real & Crespo, 2004).

The reconfiguration in hard real-time systems assumes that all incoming events, as well as their arrival rates and deadlines are known statically before system runtime. In contrast, this thesis considers systems, in which events from unknown sources can occur at arbitrary event arrival rates and have arbitrary associated deadlines.

2.2 Features of a Time-Adaptive Reconfiguration Model for Embedded Software

Based on the previous discussion, this section lists the characteristics of reconfiguration models targeting embedded software. These characteristics can be divided into characteristics that deal with the underlying system model, characteristics of the reconfiguration model itself, and characteristics of the execution model.

2.2.1 System Model Characteristics

Reconfiguration models targeting embedded software can be categorised as system-specific, domain-specific, or generic (Zhao & Li, 2007a).

System-specific reconfiguration models target software that is strongly tied to the design of the system it is executed on. This specificity of the underlying software enables the model to make assumptions about the software being changed and to manage change in a way that is optimised for a particular system (Hillman & Warren, 2004).

Domain-specific reconfiguration models are defined on system models that model software in a specific domain. Often the software follows a particular architectural style, such as the pipe-and-filter style (Shaw & Garlan, 1996). The specificity of the underlying architecture allows the reconfiguration model to omit some capabilities. For example, a reconfiguration model defined on a stateless pipe-and-filter architecture does not need state transfer capabilities and hence can omit consistency mechanisms (Zhao & Li, 2007a). These reconfiguration models are tied to software of a specific application domain, and cannot be easily transferred to software of other systems in other application domains.

Generic system models target software that can be applied in multiple application domains and a wide range of systems. Examples of generic component models include OpenCom (Coulson et al., 2008), and Think (Polakovic et al., 2006). A characteristics of these system models is that they are defined abstractly, in terms of components and connections between components. Different implementations then map these abstractions to actual systems. The abstract definition supports building composable software, independent of the target system or application. The advantage of a reconfiguration model targeting this kind of software is that it is independent of the actual embedded system used.

2.2.2 Reconfiguration Model Characteristics

Reconfiguration model features include correctness guarantees, the point in time when reconfigurations may be triggered, and the constraints on the number of reconfiguration actions and reconfigurable software entities.

A reconfiguration must ensure that it transforms the software from a correct software configuration to another correct software configuration, as otherwise the software is not functionable. A correct software configuration is defined by Goudarzi as a configuration that has the three following properties: structural integrity requirements are fulfilled, software entities are in a mutually consistent state, and state invariants are maintained (Goudarzi & Kramer, 1996). Structural integrity requirements define the dependency relationships between software entities and the way they need to be configured (Almeida et al., 2001). Structural integrity is achieved by updating the references of all entities that use a reconfigured entity. Mutual consistent state means that the execution state of a reconfigurable software entity should be the same before and after the reconfiguration takes place. Mutual consistent state can be achieved by first bringing the entities into a reconfiguration-safe state, before the actual reconfiguration is executed. However, this is an intrusive process that takes execution duration. An alternative to first bringing the entities into a mutual consistent state is to apply transitional reconfiguration that maintains the old and new system configuration in parallel, see Section 2.1.3. Maintaining state invariants is achieved by transferring state between the old and the new software entity.

Reconfiguration approaches differ in whether a reconfiguration is directly executed when it is triggered, or whether it is executed in a delayed fashion, after some other activity has finished completion. Moreover, approaches differ in whether they support an arbitrary number of entities to be reconfigured, for example by supporting the integration of previously unknown software entities, or whether they put constraints on the number of entities to be reconfigured.

2.2.3 Execution Model Characteristics

A reconfiguration that is correct and fulfils a system's constraints must reach a consistent state by either completing all its actions eventually or by undoing some of the actions, already executed. Depending on its associated execution model, reconfiguration models either react to incoming events that occur arbitrarily, and process them within their deadline, or do not consider incoming events at all.

2.2.4 Summary

This section summarises the set of characteristics of reconfiguration models targeting embedded software, and their possible values. The values, which are most likely to suit a time-adaptive reconfiguration model are shown in bold. Based on this, a set of characteristics for a time-adaptive reconfiguration model is derived.

- **System Model**
  - Entities: system-specific, domain-specific, **general**
- **Reconfiguration Model**
  - Point of time when reconfiguration starts: **immediately**, delayed

  - System correctness: **quiescent state**, **transitional approach**
  - Possible number of entities to be reconfigured: constrained, **unconstrained**

- **Reconfiguration Execution Model**
  - Guaranteed reconfiguration completion: **Supported due to transactional execution model**, not supported
  - React to previously unknown incoming events: **Supported due to preemptive execution model**, only supported for known events, not supported

These characteristics lead to the following set of required features for a time-adaptive reconfiguration model:

- **System Model**

  F1) The reconfiguration model is defined for a system model suitable for representing embedded software.

- **Reconfiguration Model**

  F2) A reconfiguration can be triggered at any time and there is no a-priori knowledge of the possible components that are to be reconfigured in the system.

  F3) The reconfiguration model does not destroy system correctness.

  F4) The reconfiguration model itself does not impose constraints on the reconfigurable entities.

  F5) There are no restrictions on the number of components to be reconfigured.

- **Reconfiguration Execution Model**

  F6) An ongoing reconfiguration is to be completed.

  F7) The reconfiguration model allows the interruption of the ongoing reconfiguration by incoming events that are previously unknown to the system.

F8) The reconfiguration model meets an incoming event's processing deadline.

These features are used in the next section to review existing reconfiguration models targeting different kinds of embedded software.

2.3 Review of Existing Reconfiguration Models

This section presents detailed reviews of reconfiguration models that target different types of embedded software, such as operating system, embedded application, and middleware software. The reviews focus on the degree to which the reconfiguration models represent the features of a time-adaptive reconfiguration model. In all of these models, reconfigurations are triggered for adaptation or evolution purposes and a central entity running on a single processor platform executes the reconfiguration sequence on the underlying software entities. Although the domain of addressed software is a super-set of the kind of software addressed in this thesis, the models include a representative sample of the techniques used to reconfigure embedded software that influenced the design of TimeAdapt.

2.3.1 Runes

Traditional middleware approaches are unsuitable for embedded systems, as they integrate all functionality that might ever be needed, resulting in memory-consuming monolithic black-box implementations. In contrast, dedicated middleware platforms for embedded systems are organised as a group of collaborating components (Kon et al., 2002). This allows the composition of very small middleware kernels that consist of only required functionality. The Runes (Reconfigurable, Ubiquitous, Networked, Embedded Systems) project developed such a middleware architecture that can target a variety of networked embedded systems, such as resource-constrained sensor nodes (Costa et al., 2007).

System Model The Runes middleware platform is component-based and encapsulates its functionality behind interfaces. The overall architecture follows a two-layer approach. The first layer comprises the basic middleware kernel that provides an API, which allows the dynamic instantiation and registration of components. The second layer comprises components that encapsulate the basic functionalities of applications and middleware services, such as measurement data collection and dissemination components on sensor nodes or publish-subscribe infrastructures on more powerful devices such as laptops.

The component model defines the general architecture of its components in an implementation-independent way in terms of the OMG's Interface Definition Language (IDL) (Object Management Group, 1999). The abstract definition of the component model allows the deployment of components on a variety of system platforms, for which implementations of the system model exist.

Reconfiguration Model The dynamic reconfiguration model is a direct implementation of Kramer and Magee's reconfiguration model with components as the underlying software entities to be reconfigured. The representation of the current system configuration as a system graph is realised by using an approach based on architectural reflection (Cazzola et al., 1998). The component model provides an architectural topology of currently installed and connected software components and interfaces describing operations to inspect and change the self-representation (also known as meta-object protocol (MOP)). Different MOPs reconfigure different parts of the components and their configuration. An architectural MOP allows the structural change of a component's composition at runtime and an interception MOP allows the behavioural adaptation of a component by introducing interceptors that can add additional behaviour. The middleware kernel represents the central configuration manager that manages and executes the reconfiguration actions on the different components.

The Runes reconfiguration model applies an approach based on quiescence to achieve mutually consistent states. This is either achieved through simple reader-writer locks or more complex algorithms that support cyclic connections between components (Rasche & Polze, 2008). Reconfiguration actions are then incrementally executed.

The reconfiguration model itself does not put constraints on the number of reconfiguration actions. It also does not put any constraints on the entities to be reconfigured or the point in time when a reconfiguration may be requested.

Execution Model The reconfiguration execution follows a transactional reconfiguration model. Reconfiguration actions are executed in an uninterruptable manner and the model does not support the interruption of an ongoing reconfiguration for incoming events.

Summary Table 2.2 summarises the features of the Rune's reconfiguration model. As can be seen from the table, the Runes reconfiguration model does not allow the immediate processing of incoming events.

Feature	System Model (F1)	F2	F3	F4	F5	F6	F7	F8
	Generic	✔	✔	✔	✔	✔	✘	✘

Table 2.2: Features of Runes reconfiguration model

2.3.2 Think

Think is an implementation of the Fractal component model targeting component-based operating systems (Polakovic et al., 2006). The approach tries to improve on existing work for reconfigurable operating system software by loosening the dependence between the reconfiguration model and the underlying platform on which the model is executed.

System Model Fractal defines a hierarchical, reflective component model. It is implementation-independent and is used for a wide range of systems, from operating systems to middleware platforms and to graphical user interfaces (OW2 Consortium, 1999). The generic system model supports the application of the reconfiguration model to many operating system platforms. The component model distinguishes two kinds of components. Primitive components can be seen as blackboxes, providing and requiring functionality through their interfaces. Composite components are composed of other components, either primitive or composite components. Each component logically comprises two parts, an internal part that implements the functional interfaces of the component and an encapsulating membrane that contains an arbitrary number of control interfaces. Control interfaces provide reflection capabilities such as the manipulation of a component's interfaces.

Reconfiguration Model Like Runes, Think's dynamic reconfiguration model is a direct implementation of Kramer and Magee's, with software components as the underlying reconfigurable entities. Reconfiguration actions in the Think model are either introspection operations, such as the reflective lookup of component interfaces, or intercession operations, such as the actual modification of the system. Reconfigurations can either be expressed directly at the level of reconfiguration primitives, provided by the Fractal API, or by using a higher-level domain-specific language such as FScript (David & Ledoux, 2006b).

The Think reconfiguration model is based on quiescence to achieve mutually consistent states. This is achieved using approaches such as thread counting or using dynamic interceptors (Polakovic et al., 2006). Reconfigurations are then sequentially executed by the root composite component, which acts as a centralised reconfiguration manager.

Execution Model The reconfiguration execution model is transactional, which implies that either a sequence of reconfiguration operations is completely executed or the system is brought back to the state it was in before the reconfiguration took place. Incoming events that coincide with an ongoing reconfiguration are not considered.

Summary Table 2.3 summarises the features of Think's reconfiguration model. Like Runes, the reconfiguration model does not allow the direct processing of incoming events and does not guarantee that an event's deadline is met.

Feature	System Model (F1)	F2	F3	F4	F5	F6	F7	F8
	Generic	✔	✔	✔	✔	✔	✘	✘

Table 2.3: Features of Think reconfiguration model

2.3.3 DynamicCon

DynamicCon extends the OSA+ middleware to support dynamic reconfiguration (Schneider, 2004). The OSA+ middleware is a middleware system for distributed, real-time embedded systems with limited memory and computational resources (Brinkschulte et al., 2000).

System Model OSA+ is a service-based middleware, with a service as the unit of reconfiguration. A service is an active entity in the middleware and can be comprised of multiple objects. Services have individual control flows to perform application or system tasks and communicate with each other by exchanging jobs. The execution of services is scheduled according to the priorities or deadlines of their corresponding jobs.

Reconfiguration Model DynamicCon has two reconfiguration action types: those that replace an old service implementation with a new service implementation or those that migrate a service to another platform. A reconfiguration is realised as a job and therefore respects real-time priorities and deadlines. Jobs with higher priorities are executed before the reconfiguration, while jobs with lower priorities are executed after the reconfiguration. Because a reconfiguration is realised as a job, a reconfiguration might not be executed directly, but instead has to wait until higher-priority jobs have completed. A service requesting a reconfiguration sends a reconfiguration request to the reconfiguration service, which then handles all reconfiguration related issues such as plugging in the new service, transferring state from the old service to the new service, and deleting the old service.

In contrast to the previously discussed reconfiguration models, DynamicCon applies a transitional approach to guarantee correctness. The new service is loaded, while the old service can still process requests. A switch between the old and the new service is executed when the complete state has been transferred between the two (stateful) services. One of the main goals of the reconfiguration model is to minimise the blackout time in which a service that is currently reconfigured is unavailable. The model supports different reconfiguration approaches that have different blackout time durations. In the *full blocking approach*, the new service is blocked until the old service explicitly calls the switch statement to initiate the exchange of both services. After the switch is executed, the state is transferred. This causes the longest blackout time but at the same time ensures consistent and identical state information on both sides. In the *non-blocking approach*, the new service version starts to transfer state. During reconfiguration, if the remaining state between the old and the new service can be transferred during this requested blackout time, the reconfiguration is executed completely and all new incoming jobs are processed by the new service. Otherwise, if the requested blackout time cannot be met, the reconfiguration is interrupted and the reconfiguration service keeps monitoring changes to the service state. Incoming jobs are then processed by the old service. Figure 2.2 illustrates this principle.

Execution Model The reconfiguration model takes a transactional view on reconfiguration execution. Incoming events, represented as jobs, are processed after the reconfiguration service returns. The reconfiguration model does not support the direct preemption of a reconfiguration to an incoming event. Using the non-blocking approach, a reconfiguration returns if the specified blackout time cannot be met, which is the minimum time before a job with a higher priority should be processed. If this job represents

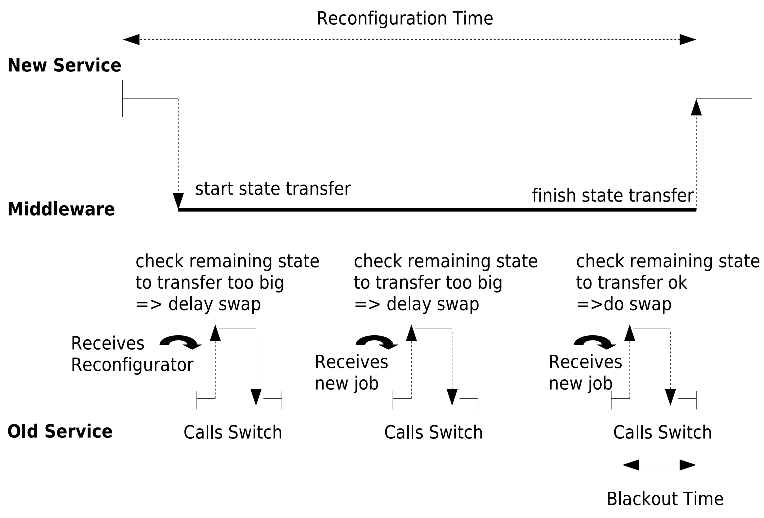


Fig. 2.2: The non-blocking reconfiguration approach (Schneider, 2004)

an event, its response deadline will be met only if it is larger than the requested blackout time.

Summary Table 2.4 summarises the features of DynamicCon's reconfiguration model. The constraints on the start time of a reconfiguration and the constraints on support for incoming events makes this reconfiguration model not the ideal candidate for realising a time-adaptive reconfiguration model.

Feature	System Model (F1)	F2	F3	F4	F5	F6	F7	F8
	System-specific	✘	✔	✔	✔	✔	✘	✘

Table 2.4: Features of DynamicCon reconfiguration model

2.3.4 DynaQoS-RDF

The DynaQoS-RDF reconfiguration model targets dataflow-driven embedded application software (Li, 2009). Its main aim is to minimise the effect an ongoing dynamic reconfiguration has on system performance, such as throughput or response time. The reconfiguration model has no application disruption time because instead of waiting until a reconfiguration-safe state is reached, the model maintains the old and the new system configuration in parallel.

System Model The reconfiguration model is defined on an abstract system model that follows a dataflow-driven computational model, the reconfigurable dataflow system model (RDF) (Li, 2009). It allows the modelling of a wide range of embedded software, such as signal processing software, linear and non-linear control system software, and image processing software. The system model extends the concept of a dataflow process network with reconfiguration capabilities of its basic elements, such as processes, data-stores and data-paths (Lee et al., 2003). A process is a computational entity that consumes data through its input ports, processes them, and produces results through its output ports. A data-store holds a specific amount of data, whereas a data-path connects a process with a data-store. Two processes communicate indirectly with each other via their connected data-store.

Reconfiguration Model The reconfiguration model supports the addition and removal of processes and data-stores, as well as the addition and removal of data-paths. The model follows a transitional approach to realise consistent system transformations. A dynamic reconfiguration in this model is divided into three phases. Phase one establishes the new configuration. In this phase, software entities, which were previously not installed, are loaded into memory. New incoming data uses this new configuration. Phase two handles switching between the old and the new configuration. Switching includes the completion of data processing by system entities that are part of the old configuration, and starting the processing of new data items only by entities that are part of the new configuration. Phase three removes all software entities that are no longer in use.

The parallel execution of the two configurations leads to a number of issues that are solved by the reconfiguration model, such as guaranteeing transactional non-interleaving and transactional completeness. A transaction is a logical concept that refers to the processing of a requested data-item until the corresponding result is generated. Transactions comprise multiple processes and their connections via data-paths. Transactional interleaving occurs if transactions that belong to different configurations interfere. The reconfiguration model avoids these interleavings by applying a version control mechanism that isolates data flows belonging to different configurations. Every data-item is assigned a version tag and a process can decide to process this data-item, based on its version number. Transactional completeness means that a transaction, once started, should be completed. This is particularly important during the shutdown period, so that all transactions using processes or data-stores to be removed, are completed. Transactional completeness is supported by the underlying system model, in the form of synchronisation mechanisms and a tracing mechanism that makes sure that a processor or data-store/path is not currently used so that it can be removed safely.

A reconfiguration can be requested at any given time, and the model supports the introduction of previously unknown software entities and the reconfiguration of an arbitrary number of software entities. This puts constraints on the actual entities to be reconfigured. As a pre-condition, each reconfiguration must be directly interruptible. This means that the reconfiguration model does not support reconfiguration actions that will last for a period of time and have an influence on the actual system configuration, such as state transfer. Hence, only stateless software entities are supported.

Execution Model The reconfiguration execution model can be classified as preemptive, as it interrupts the reconfiguration for incoming events, which are associated with a higher priority. A reconfiguration can be interrupted at any given point in its execution. If the event arrival rate is high, a reconfiguration completion might be delayed indefinitely.

Summary Table 2.5 summarises the features of the DynaQoS-RDF reconfiguration model. The reconfiguration model has some intrinsic features of a time-adaptive reconfiguration model, namely its execution model allows the reaction to arbitrary incoming events. However, in the case of many incoming events, the completion of a reconfiguration cannot be guaranteed. Also, the model itself imposes constraints on the reconfigurable entities, such as only the support of stateless software entities.

Feature	System Model (F1)	F2	F3	F4	F5	F6	F7	F8
	Generic	✔	✔	✘	✔	✘	✔	✔

Table 2.5: Features of DynaQoS-RDF reconfiguration model

2.3.5 Djinn

Djinn is a programming framework, which supports the construction and dynamic reconfiguration of distributed multimedia applications targeting embedded systems, such as digital television production, security, and medical systems (Mitchell et al., 1999).

System Model The Djinn system model is generic. Similar to the RDF system model, a multimedia application comprises a set of active components, which consume, transform, and produce media data streams and which are connected via ports. However, the framework uses a split-level component architecture; components in the framework are either model or peer components. Peer components are active objects that are potentially distributed across multiple hosts and that have associated temporal constraints on their operations. They are also the unit of reconfiguration. Model components abstract the functionality of the peer components and emphasise the QoS characteristics of their underlying peer components. They support the separation of the application design from its runtime realisation.

Reconfiguration Model Reconfigurations can be required at any point in time, even though it depends on the actual scheduling time as to when a reconfiguration action will manifest itself in the system. Reconfigurations are executed at the model component layer, with their effects made visible at the peer layer when a reconfiguration is successfully completed. The execution of the reconfiguration on the model level allows the system to validate the reconfigurations with regards to structural and data constraints. A reconfiguration is expressed in terms of paths. A path encapsulates a group of ports and active components that carry its data. Associated with each path are QoS properties such as latency, jitter and error rate. A reconfiguration will always replace an entire path or sub-path with a new one and has actions such as creating a new active component, deleting active components, and switching input and output ports to the respective active component to use. Figure 2.3 illustrates a reconfiguration example, in which a single video unit is reconfigured to use dial up GSM links instead of local WLan links.

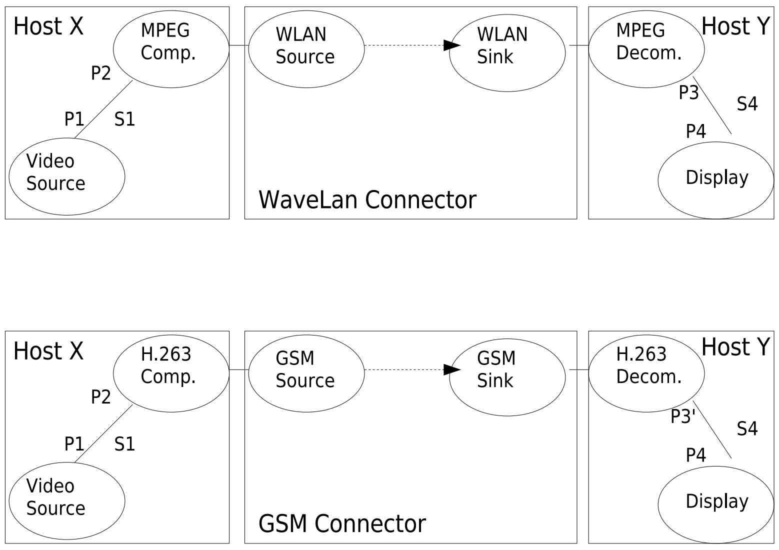


Fig. 2.3: Reconfiguration of a single mobile unit to different communication mechanisms (Mitchell et al., 1998)

Reconfiguration consistency is achieved by using a transitional approach, i.e., switching between the old and the new configuration. One of the main challenges is maintenance of temporal properties during reconfiguration, such as the maximum arrival rate allowed between two data items. Meeting these properties requires reconfiguration action scheduling. The scheduling algorithm determines when to send events that trigger the activation of a reconfiguration action, so that temporal properties are not violated. To avoid a startup delay for newly integrated components, the reconfiguration is divided into two phases, namely a setup and an integration phase. In the setup phase, constraints on the new configuration are checked and new peer components are created as well as resources reserved. The integration phase completes the transition to the final configuration by applying the computed schedule. As illustrated in Figure 2.3, this means that the start event, received by input port P2' of the H.263 component, needs to be injected before the stop event, received by input port P2 of the MPEG compression component, is sent. This schedule ensures that frames arrive simultaneously at the input port P4 of the display component.

Execution Model The reconfiguration execution model is transactional. This implies that if not all of the required reconfiguration actions can be scheduled successfully, then none of the actions will be performed and the application will remain in its initial state. The main aim of the model is to maintain application timeliness properties during reconfiguration. The processing of incoming events during a reconfiguration is not supported and a timely response to their deadline not ensured.

Summary Table 2.6 summarises the features of the reconfiguration model. Djinn follows the same execution model as Runes and Think and lacks support for reacting to incoming events and processing them within their event deadlines. Also, the model requires a-priori knowledge of the components to be reconfigured, as it only reconfigures components that are part of an existing path.

Feature	System Model (F1)	F2	F3	F4	F5	F6	F7	F8
	Generic	✘	✔	✔	✔	✔	✘	✘

Table 2.6: Features of Djinn reconfiguration model

2.3.6 Port-based Objects

The Port-based Object (PBO) abstraction supports the design and implementation of dynamically reconfigurable real-time control software (Stewart & Khosla, 1996), which can be executed in single and multiprocessor environment. Unlike the previously discussed reconfiguration models, dynamic reconfiguration in this model refers not to the structural change of the system topology but rather switching on and off software entities as part of the current configuration.

System Model Properties A PBO is an independent concurrent process, which communicates with other PBOs only indirectly through its input and output ports. Figure 2.4 illustrates three PBOs A, B and C with input and output ports. The output port j of PBO B is connected with the input port j of PBO A, the output port k of PBO A is connected to input port k of PBO B and C. The output port l of PBO C is not connected to any PBO. This loosely coupled infrastructure allows easy replacement of PBOs and makes them the unit of reconfiguration.

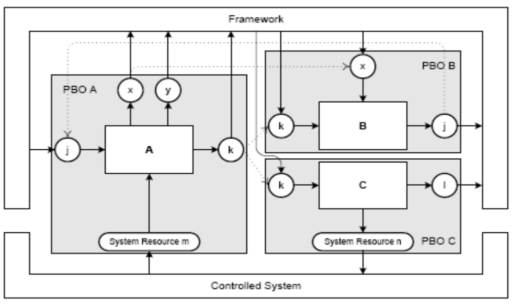


Fig. 2.4: Communication between PBOs (Issel, 2006)

The framework is executed on a controlled system environment, the Chimera RTOS, that provides well-defined communication and memory mechanisms (Stewart et al., 1992).

Reconfiguration Model The dynamic reconfiguration model is strongly tied to the underlying system model, making it a system-specific reconfiguration model. In contrast to structural changes, which change the topology of the current software, and implementation changes, which change the internals of a software entity, this reconfiguration model changes only the system state of PBOs. All PBOs that are in the *ON* state denote the currently active configuration. A reconfiguration transforms the currently active configuration to a new active configuration. All PBOs that are part of the new configuration and that are in the *OFF* state are transformed to the *ON* state and vice-versa.

The reconfiguration steps themselves are executed in idle times of the framework scheduler, i.e., when no PBO is currently executing. An executing PBO is always executed until its completion so that there is no need for a mechanism to explicitly drive the PBOs into a reconfiguration-safe state.

The model imposes constraints on the number of entities to reconfigure, as it can switch a maximum of n entities, with n denoting the maximum configuration. Entities must be realised by the underlying framework that imposes constraints on those to be reconfigured. Also, switching requires that all modules, potentially required in some configuration, need to be known beforehand. This excludes the integration of new software modules downloaded from other systems or the environment after system startup.

Execution Model The reconfiguration execution model follows the preemptive approach as the underlying scheduling mechanism is priority-based. Higher-priority PBOs can interrupt the ongoing reconfiguration. However, the model is very limited, in that it interrupts only for PBOs that are statically known before any reconfiguration takes place and not to incoming events from sources, which were unknown at system design time. Because PBOs are scheduled statically, worst-case execution durations for a reconfiguration can be calculated. The reconfiguration time t_r is the sum of the duration of setting the off and on methods of affected PBOS, the duration of setting up output ports, and the duration of PBOs that interrupt the reconfiguration due to their higher priority. Static scheduling does, however, guarantees reconfiguration completion.

Summary Table 2.7 summarises the features of the PBO reconfiguration model. Because all PBOs are scheduled statically, a reconfiguration cannot be requested at any time, and the number of reconfigurable entities is restricted by the size of the maximum possible configuration of PBOs. PBO's system model is specific to a platform and imposes restrictions on the entities, as only entities that follow this model can be reconfigured. The reconfiguration model does not react to incoming events that are unknown at system design time. However, the static scheduling of PBOs guarantees eventual reconfiguration completion and the meeting of timeliness requirements of other PBOs.

Feature	System Model (F1)	F2	F3	F4	F5	F6	F7	F8
	System-specific	✘	✔	✘	✘	✔	✘	✔

Table 2.7: Features of PBO reconfiguration model

2.3.7 Adaptive Reconfiguration Models

The previously discussed reconfiguration models conform to the blackbox philosophy, as they encapsulate a single and fixed reconfiguration process. This section discusses reconfiguration models that apply a non-static, i.e., adaptive, reconfiguration process. Adaptive in this context does not mean the reason for reconfiguration is adaptive, but rather that the reconfiguration process itself can be changed. Current adaptive reconfiguration models enable the customisation of a reconfiguration process depending on the underlying properties of the entities to be reconfigured. However, the adaptation occurs before the reconfiguration process starts. Once a customised reconfiguration process runs, it is executed following a transactional or preemptive execution model. In contrast to these reconfiguration processes, a time-adaptive reconfiguration process that reacts to incoming events needs to be flexible in terms of which reconfiguration actions still to execute, in order to meet the event's response deadline and to prevent reconfiguration starvation. Such a reconfiguration process needs dynamic adaptation based on the current conditions, such as time constraints, and the entities to be reconfigured.

This section describes two adaptive reconfiguration models that target embedded software, such as operating system software and network stack software. The first is called Molecule and it targets resource-constrained sensor operating systems and selects an appropriate linking method of the modules (Yi et al., 2008). The second, NecoMan, is a middleware that supports the dynamic reconfiguration of programmable network services and customises the applied reconfiguration process according to the properties of the current services that are reconfigured (Janssens et al., 2005).

2.3.7.1 Molecule

Molecule is an adaptive reconfiguration model that targets resource-constrained sensor operating systems (Yi et al., 2008).

System Model In Molecule, software entities are managed as modules. Modules are connected with each other via linking mechanisms. There are two types of linking mechanism: direct linking and indirect linking. In direct linking, the calling address of the function to be called is directly linked with the actual function. In indirect linking, the address of the function is obtained by first accessing a global function table. These linking mechanisms have different execution and reconfiguration times. For example, the direct linking mechanism is faster in terms of execution duration, but results in higher reconfiguration costs.

Reconfiguration Model Reconfiguration in Molecule targets the linking between modules, such as relinking to a different module or linking to a new module. When a module is reconfigured, the reconfiguration model first checks which linking method is better suited for required services from other modules. The appropriate linking method is based on the cost analysis of the expected timing overhead of each module when using the different linking mechanisms. The reconfiguration also dynamically updates either the call address of all modules that link to the reconfigured module when using direct linking, or updates the global function table.

The system assumes a consistent software configuration, as reconfigurations are only executed in idle times, when no module is active.

Execution Model Reconfigurations are executed in a transactional manner, without the consideration of incoming events.

Summary Table 2.8 summarises the features of the reconfiguration model. The execution model applied is the same as for Runes and Think, and does not consider incoming events and their associated time deadlines. The model itself has no restrictions on the number of reconfigurable entities, or when a reconfiguration can be executed. However, the reconfiguration is very specific to the system model used as it only addresses relinking of modules.

Feature	System Model (F1)	F2	F3	F4	F5	F6	F7	F8
	System-specific	✔	✔	✘	✔	✔	✘	✘

Table 2.8: Features of Molecule reconfiguration model

2.3.7.2 NecoMan

NecoMan is a middleware for programmable networks (Janssens et al., 2005). Its reconfiguration model realises distributed reconfigurations, which includes the coordination of reconfiguration actions among different nodes.

System Model The reconfiguration model is defined for a domain-specific system model that targets reconfigurable network stack software (Michiels, 2003). The DIPS+ component model adopts the pipe-and-filter architectural style. DIPS+ components are two-layered, with a core containing the basic functionality, and communication ports for connecting to other components. All incoming ports share the same interface, which allows a simplified composition without the need to check for compatibility when connecting components to each other.

Reconfiguration Model The middleware has been developed to improve the effectiveness of the reconfiguration process applied. This process is customised based on the network service that will be deployed and the underlying execution environment. The reconfiguration model follows a reconfiguration process that is divided into three steps:

1. The installation of a new service by extending the programmable nodes targeted for service deployment with new sending and receiving modules.

2. The deactivation of the old service by waiting until all processing of data has completed. .

3. The activation of the new service by removing the old service and updating the connections of other services to point to the new service.

The reconfiguration process can be changed by exploiting the properties of the network service to be deployed and the underlying execution model. For example, when the service is stateless, there is no need to bring the service into a reconfiguration-safe state. In this case, a transitional approach can be applied, in which the new service is concurrently deployed with the old service. This implies that the activation and the finishing phases are switched (activate before finish). Other optimisations are possible that are not relevant to this discussion (Janssens, 2006).

The reconfiguration model itself imposes no constraints on the time when a reconfiguration takes place, or the number of entities that are to be reconfigured.

Execution Model The execution model of NecoMan follows that of other transactional reconfiguration models, such as the Runes' or Think model. A sequence of reconfiguration actions is either executed fully, or not at all. Incoming events are not considered.

Summary Table 2.9 summarises the features of the reconfiguration model. Like the Runes or Think reconfiguration models, the model does not react to incoming events and their deadlines and cannot be classified as time-adaptive.

Feature	System Model (F1)	F2	F3	F4	F5	F6	F7	F8
	Domain-specific	✔	✔	✔	✔	✔	✘	✘

Table 2.9: Features of NecoMan reconfiguration model

2.4 Analysis

The previous section reviewed existing work on dynamic reconfiguration of embedded software. The commonality of these approaches is that reconfigurations are executed by a centralised entity on the componentised software entities, for adaptation or evolution purposes. This section provides an analysis of these reconfiguration models with respect to the degree to which they support the requirements identified in Section 2.2. Where appropriate, the corresponding feature number is added, in parentheses, to the respective criterium. Features F2, F4, and F5 are grouped for this section into their own comparison group, as they represent constraints of either the underlying software or the reconfiguration model itself.

2.4.1 System Model Requirements

The first set of features considered are the unit of reconfiguration dictated by the underlying system model (F1), and whether the system model is system-specific, domain-specific, or generic (see Table 2.10).

Reconfiguration models that target real-time embedded software, such as the reconfiguration model for Port-based Objects (PBO) or DynamicCon, are system-specific and hence strongly tied to a specific platform on which the software runs on. As this software is often safety-critical, the reconfiguration process itself needs to be verifiable (Trapp et al., 2007). The system model itself can provide guarantees for the reconfiguration model. For example, the controlled execution system in the PBO reconfiguration model allows known worst-case execution durations (Stewart & Khosla, 1996). However, in system-specific reconfiguration models, the tight integration of the addressed software with the underlying embedded platform makes it hard to leverage the model on other embedded platforms. As we target a time-adaptive reconfiguration model for various embedded platforms and their software, this kind of system model is not suitable.

The NecoMan reconfiguration model is a domain-specific system model, as it has been defined for the DIPS component model, a software architecture that is tailored towards building reconfigurable protocol stacks (Michiels, 2003). Its underlying software system is optimised for a particular domain and lacks generality, as it cannot model general embedded software.

In contrast, Runes, Think, DynaQoS-RDF, and Djinn are defined for system models that can model various embedded software. Generality is achieved by defining reconfiguration operations on an abstractly defined system model. A specific implementation then realises the abstract defined operations and software entities for a specific embedded software, which can be executed on multiple embedded platforms. For example, Think is a C-based implementation of the Fractal architecture, realising component-based operating system kernel software (Polakovic & Stefani, 2008).

Reviewed Model	Unit of Reconfiguration	Generic (F1)
Runes	Flat Reflective Component	✔
Think	Hierarchical Reflective Component	✔
DynamicCon	Stateful& Stateless Service	✘ System-specific
DynaQoS-RDF	Stateless Reconfigurable Data-Flow System Entity	✔
Djinn	Multimedia Components	✔
PBO	Port-based Objects	✘ System-specific
Molecule	Module	✘ System-specific
NecoMan	Flat Reflective Component	✘ Domain-specific

Table 2.10: Comparison of system model properties

2.4.2 Reconfiguration Model Features

This section considers features specific to the reconfiguration model. These include the mechanism used to reach a consistent state (F3), and its underlying execution model (see Table 2.11).

All of the reviewed reconfiguration models ensure system correctness. Molecule and PBO can assume a consistent system as they execute reconfigurations only in idle times, when no software entity is active. These reconfiguration models do not, therefore, need to explicitly drive entities into mutually consistent states. Runes, Think, and NecoMan provide consistency based on quiescence. They first bring the affected system entities into a reconfiguration-safe state and then execute the sequence of reconfiguration actions. DynamicCon, DynaQoS-RDF, and Djinn provide consistency based on transitioning between the old and the new system configuration. These models first execute a sequence of reconfiguration actions concerned with building up the new system configuration. They then switch to the new system configuration, once the state has been completely transferred and all ongoing processes that use the old configuration's system entities have been completed.

With the exception of DynaQoS-RDF and PBO, all of the reviewed reconfiguration models have a transactional execution model. They guarantee the completion of a reconfiguration if the reconfiguration is structurally correct and transform the system into a consistent state.

DynaQoS-RDF follows a preemptive reconfiguration execution model and can preempt an ongoing reconfiguration to previously unknown incoming events that have a higher priority. The PBO reconfiguration model only pre-empts an ongoing reconfiguration to statically known, higher-priority events.

Reviewed Model	Consistency Mechanism (F3)	Reconf. Execution Model
Runes	Quiescence	Transactional
Think	Quiescence	Transactional
DynamicCon	Transitional	Transactional
DynaQoS-RDF	Transitional	Pre-emptive
Djinn	Transitional	Transactional
PBO	N/A	Pre-emptive
Molecule	N/A	Transactional
NecoMan	Quiescence	Transactional

Table 2.11: Comparison of reconfiguration model properties

2.4.3 Execution Model Features

This section considers the features specific to the execution model. These include whether a reconfiguration completeness is guaranteed (F6), whether arbitrary, i.e., unknown, incoming events are supported (F7), and whether an incoming event's response deadline is met (F8) (see Table 2.12).

All of the reviewed reconfiguration models, except DynaQoS-RDF, can guarantee the completion of an ongoing reconfiguration. DynaQos-RDF cannot guarantee the completion of an ongoing reconfiguration, when there is a high event arrival rate. However, the model directly interrupts an ongoing reconfiguration to process incoming events, and hence meets the processing deadlines of these events. The remaining reconfiguration models meet an incoming event's deadline only, if the completion time of a reconfiguration is shorter than the response deadline. For example, using a non-blocking approach, the DynamicCon reconfiguration model returns from the reconfiguration process if the remaining state of an entity cannot be transferred within the requested blackout time. The deadline of a subsequent job can be met if a job's response time is larger than this requested blackout time and the job's processing time. However, this approach does not support arbitrary events but only jobs that are triggered by a communication with other services or by the system itself.

Reviewed Model	Reconfiguration completion (F6)	Reaction to incoming events(F7)	Meeting Deadlines (F8)
Runes	✔	✘	✘
Think	✔	✘	✘
DynamicCon	✔	✘	✘
DynaQoS-RDF	✘	✔	✔
Djinn	✔	✘	✘
PBO	✔	✘	✘
Molecule	✔	✘	✘
NecoMan	✔	✘	✘

Table 2.12: Comparison of execution model features

2.4.4 Reconfiguration Constraints

Table 2.13 compares the reconfiguration models in terms of their constraints, either on the number of entities to be reconfigured (F5), the entities themselves (F4), or the reconfiguration process (F2). Runes and Think have no constraints on the number of entities that can be reconfigured and do not put constraints on the entities or the reconfiguration model itself. Both models support the reconfiguration of stateless and stateful components, as well as the reconfiguration of any component in the system graph. For example, in Think a hierarchical component can contain any number of sub-components that all can be potentially reconfigured.

DynamicCon does not put any constraints on the number of system entities to be reconfigured. However, reconfigurations cannot be executed at arbitrary times. Since a reconfiguration is realised as a job, maintaining job priorities and deadlines, a reconfiguration might not be executed directly, but instead has to wait until higher-priority jobs have completed. Also, the system-specific system model requires that reconfigurable software entities are realised as services.

DynaQoS-RDF can potentially reconfigure the overall system. Its pre-condition that a reconfiguration always must be directly pre-emptable means that the model supports only stateless system entities and puts constraints on the entities itself. The model supports the integration of previously unknown entities in the system.

Djinn's reconfiguration model does not constrain the number of reconfiguration actions. However, it constrains the underlying system entities as they must adhere to the software architecture of the DJINN programming model. Though reconfigurations can be requested at any point in time, there might be a delay in when they are actually applied.

The PBO reconfiguration model can reconfigure only the system entities that are available in the system at system startup time, as a reconfiguration consists of switching on and off PBOs. Entities must follow a domain-specific framework, namely the Port-based object abstraction and specific communication mechanisms, such as local and shared memory with bounded access times. There are strict scheduling requirements in the framework and to guarantee a timely execution of all PBOs and the reconfiguration itself, all configurations must be known statically at system startup time.

The NecoMan reconfiguration model does not impose any constraints on the number of entities, or the underlying reconfiguration model. However, this reconfiguration model is defined on a domain-specific system model for reconfigurable network protocol stacks and cannot be transferred to other embedded system software.

The same arguments hold for the Molecule reconfiguration model. Molecule can reconfigure an optional number of modules and does not put any constraints on the modules themselves. However, it concentrates on providing efficient linking mechanisms and does not provide structural and functional reconfiguration. To use these linking mechanisms, the software entities must be realised with the system-specific system model.

Reviewed Model	No constraints on number of entities (F5)	No entity constraints (F4)	No reconfiguration constraints (F2)
Runes	✔	✔	✔
Think	✔	✔	✔
DynamicCon	✔	✘	✘
DynaQoS-RDF	✔	✘	✘
Djinn	✔	✘	✘
PBO	✘	✘	✘
Molecule	✔	✘	✘
NecoMan	✔	✘	✔

Table 2.13: Comparison of constraint properties

A summary of all comparisons is shown in Table 2.14. As illustrated, none of the reviewed systems has all the features that are required by a time-adaptive reconfiguration model.

Reviewed Model	F1	F2	F3	F4	F5	F6	F7	F8
Runes	✔	✔	✔	✔	✔	✔	✘	✘
Think	✔	✔	✔	✔	✔	✔	✘	✘
DynamicCon	✘	✘	✔	✘	✔	✔	✘	✘
DynaQoS-RDF	✘	✘	✔	✘	✔	✘	✔	✔
Djinn	✘	✘	✔	✘	✔	✔	✘	✘
PBO	✘	✘	✔	✘	✘	✔	✘	✘
Molecule	✘	✘	✔	✘	✔	✔	✘	✘
NecoMan	✔	✔	✔	✘	✔	✔	✘	✘

Table 2.14: Comparison summary

2.5 Relevance to TimeAdapt

This section discusses concepts from the reviewed reconfiguration models that have been influential on the design of the time-adaptive reconfiguration model, TimeAdapt, discussed in this thesis.

2.5.1 System Model

Reconfiguration models can be defined either on a specific or an abstract system model. Abstractly defined system models define their associated entities and the connection between those entities in an implementation-independent way. The abstract definition hides the complexities of an underlying system platform and focusses rather on characteristics of the entities themselves, such as their underlying computational model. Different implementations of the abstract definitions for different embedded system platforms, allows the system model to be executed on multiple embedded platforms. TimeAdapt is defined on such an abstract system model. A more detailed description of TimeAdapt's system model and the rationale of its use is given in chapter 3.2.1.

2.5.2 Reconfiguration Model

All the reviewed reconfiguration models ensure system correctness by either using an approach based on quiescence or a transitional approach. An approach based on quiescence drives the system into a reconfiguration-safe state before the actual reconfiguration is executed. A reconfiguration then incrementally transforms the existing system configuration. A transitional approach executes the old and the new system configuration in parallel and switches over between the two configurations at some stage. A more detailed discussion of the approach adopted in TimeAdapt is presented in Section 3.2.2.1.

TimeAdapt also adopts a centralised reconfiguration model architecture, similar to the reviewed systems, in which a centralised entity (reconfiguration manager) has access to all system entities and is responsible for their reconfiguration. As a time-adaptive reconfiguration model needs to consider incoming events that coincide with an ongoing reconfiguration, TimeAdapt reacts as fast as possible to these events. However, in contrast to the reviewed reconfiguration models that use an underlying priority-based scheduling mechanism, it does not directly switch to the interrupting event, but tries to execute as much reconfiguration actions as possible before handling the event. Section 3.2.2.2 discusses in more detail the applied scheduling mechanism.

2.5.3 Other Influential Concepts

TimeAdapt guarantees a correct system at all times, even in the presence of partially executed reconfigurations. Inspired by the work of Zhang et al. (2005), the model applies a partitioning of the remaining reconfiguration actions into so-called *safe steps*. Safe steps transform a safe system configuration to a new safe system configuration. Safe system configurations are defined as system configurations in which all dependency relations between the components taking part in the configuration are fulfilled so that the system can operate correctly. This is discussed further in Section 3.2.2.3.

2.6 Summary

This chapter has presented a number of reconfiguration models targeting different kinds of embedded software, such as operating system software, middleware software, and application software. In all these models, reconfigurations are triggered for adaptation and evaluation purposes and executed by a centralised entity. In particular, the models were investigated with regards to the system model their software follows, the features of the reconfiguration model itself, and the execution model the reconfiguration process follows. Based on this investigation, the features that a time-adaptive reconfiguration model should have are listed. These features were the criteria used to compare the related models. The comparison demonstrates that none of the existing work has all the features a time-adaptive reconfiguration model needs. Specifically, the models fall short in guaranteeing reconfiguration completion, while reacting to incoming events with associated processing deadlines. Finally, concepts from the reviewed existing models that have influenced the design of TimeAdapt were discussed. The following chapters detail the design, implementation, and evaluation of TimeAdapt.

Chapter 3

TimeAdapt Design

Analysis of existing reconfiguration models for embedded software shows that they are limited in their support for a timely reaction to incoming events, while making progress towards reconfiguration completion. This chapter first derives a set of requirements for a time-adaptive reconfiguration model (RM) targeting embedded software. The remainder of the chapter provides a detailed description of the design and the system elements of TimeAdapt, a time-adaptive reconfiguration model, that satisfies these requirements.

3.1 Requirements for a Time-Adaptive RM

The previous chapter discussed the limitations of existing reconfiguration models in terms of their applied execution model. A rationale was given for the development of a time-adaptive reconfiguration model that supports reaction to incoming events during an ongoing reconfiguration, while making progress with an ongoing reconfiguration. Designing a time-adaptive reconfiguration model raises numerous issues. These issues can be categorised into issues related to the system model used, issues related to the correctness of a reconfiguration, issues related to the actual execution of a reconfiguration in the presence of incoming events, and issues related to the restrictions of the reconfiguration model or system model on the entities to be reconfigured.

The selection of a suitable system model is a critical element for the design of a reconfiguration model, as it determines the potential types of reconfiguration actions as well as the degree to which the reconfiguration model can be transferred to different system platforms (Wermelinger, 1997). This observation leads to the first requirement:

R1) The unit of reconfiguration should not be system platform specific, but it should be executable on various system platforms.

Every reconfiguration model needs to ensure that the system is in a consistent state, before a reconfiguration and after completing a reconfiguration, as a transformation to an incorrect system configuration leads to erroneous system behaviour (Janssens, 2006). For this, two preconditions must be preserved; transactional completeness and the satisfaction of structural integrity requirements (Goudarzi, 1999). These are discussed in more detail in Sections 3.2.2.1 and 3.2.2.2.

R2) The reconfiguration model should ensure system correctness during and after the reconfiguration.

A time-adaptive execution model requires that a reaction to incoming events occurs in a timely manner. However, the completion of an ongoing reconfiguration needs to be guaranteed. This leads to the formulation of the following requirements for a time-adaptive reconfiguration model.

R3) An ongoing reconfiguration, that is structurally correct and fulfills the system's constraints, should be guaranteed to complete.

R4) The reconfiguration model should allow the interruption of the ongoing reconfiguration by incoming events that are previously unknown to the system.

R5) The reconfiguration model should try to process incoming events within their associated processing deadline.

Tešanović et al. (2005) proposed additional, non-functional, requirements for a reconfiguration framework that reconfigures real-time embedded systems. These non-functional concerns address the number of components supported and the point in time when a reconfiguration should be executed.

R6) There should be no restriction on the number of components to be reconfigured.

R7) There should be no constraints on the entities to be reconfigured.

R8) Reconfiguration may be requested at any time, and the system should have no a-priori knowledge of the possible components that are to be reconfigured in the system.

3.2 TimeAdapt

This section uses the requirements R1-R8 to derive the design of TimeAdapt, a time-adaptive reconfiguration model proposed by this thesis. TimeAdapt's design is broken down into the design of two orthogonal, but complementary parts: the provision of a suitable underlying system model and the design of TimeAdapt itself, which executes the modifications on the system model (see Figure 3.1).

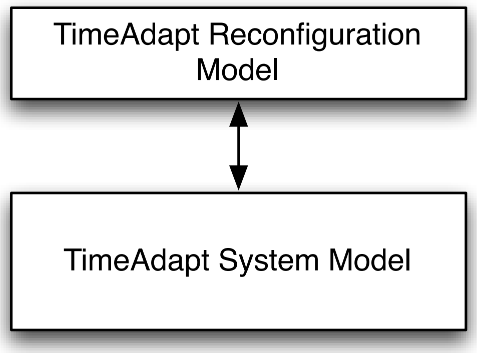


Fig. 3.1: TimeAdapt Design

3.2.1 TimeAdapt System Model

A system model, in reconfiguration literature, defines what constitutes a software entity and determines the granularity at which reconfigurations can be performed, the so-called unit of reconfiguration. A unit of reconfiguration is defined by Janssens (2006) as:

"the most fine-grained software abstraction that is subject to change."

In the review of existing reconfiguration models for embedded software in the previous chapter, we identified three main types of system models (see Section 2.2.1). These system models differ in the level of granularity of their software entities, and in the extent to which they can be applied on different embedded system platforms.

1. System-specific models, e.g., (Stewart & Khosla, 1996)
2. Domain-specific models, e.g., (Mitchell et al., 1998)
3. Abstract system models, e.g., (Costa et al., 2007)

A system-specific model defines entities that are strongly tied to a specific system platform. Examples for such software entities are software modules of a particular operating system, such as the SOS operating system for sensor nodes (Han et al., 2005). The granularity of these software entities is well-defined. However, due to the dependency of the entities on their respective platform, the applicability level of a reconfiguration model defined for this kind of system model is low. Defining TimeAdapt for this system model would imply that the reconfiguration model could only be applied on a single specific platform, which contradicts requirement R1.

A domain-specific system model defines entities that can be applied in a variety of application scenarios in a particular domain. Prominent examples are domain-specific component frameworks that allow the construction of programmable network stacks, such as Necoman (Janssens et al., 2005). The granularity of the software entities is dependent on the specific application domain. The applicability level of a reconfiguration model defined on this kind of system model is higher than in the system-specific model case, but constrained to a single domain.

A generic system model provides well-defined but abstract software entities. Such abstractions support a higher level of granularity for software entities than is possible to achieve in system-specific or domain-specific models. For example, a hierarchical component in a generic hierarchical component model can contain potentially any number of components. Generality is achieved by providing implementations of the abstractly defined software entities for respective system platforms and application domains. For example, the Runes component model API is defined in terms of the OMG's interface definition language IDL (Object Management Group, 1999). Implementations of the API map the abstractly defined concepts to actual platforms, such as implementations for resource-scarce embedded systems running on the Contiki OS (Dunkels et al., 2004) or Java-based implementations that target more resourceful platforms (Costa et al., 2007). Reconfiguration models defined on these kinds of system models can be applied on a variety of platforms and for a variety of application domains, fulfilling requirement R1. The design decision in this thesis is that TimeAdapt's system model is defined on such a generic system model.

The abstract system models used by Runes and Think, follow a synchronous computational model based on message passing between software entities, whereas the DynaQoS and Djinn reconfiguration models follow an asynchronous computational model, based on dataflow between software entities (see Chapter 2, Section 2.3). A computational model based on message passing leads to the blocking of a calling process, while waiting for a message's reply. This is less suitable for embedded software, in which several activities interact with each other, and in which a blocked activity can lead to the blocking of other activities (Lee, 2000). Therefore, TimeAdapt's system model is based on the asynchronous reconfigurable data flow system model, RDF. The RDF is an extension to the conceptual actor model (Agha, 1986). There are two characteristics of the RDF model that emphasise its usefulness for embedded software and motivate its use as TimeAdapt's system model. Firstly, the underlying mathematical abstractions of the RDF, namely actors, are well-suited to describe embedded software and concurrent software, in which parallel activities interact with each other and with the external environment, often under timing constraints (Lee, 2002). Secondly, the RDF model represents the kind of software that is targeted by the time-adaptive reconfiguration model discussed in this thesis. For scoping reasons and to concentrate on the time-adaptive reconfiguration model rather than on the system model itself, the software considered in this thesis runs on a single processor embedded platform that is subject to events associated with deadlines. Examples of this kind of software are signal or sensing applications on a single sensor node (Cheong, 2007). In contrast to the DynaQos system model, TimeAdapt's system model supports stateless and stateful entities, and in contrast to the Djinn reconfiguration model,

We consider the concept of actors from Lee (2002), who defines them as concurrent dataflow-driven components that specify behaviour in an abstract way.

Definition 1 *An actor a is a computational entity that receives (data) tokens via input ports I_a, processes data internally, and sends out (data) tokens via output ports O_a.*

A port represents a connection to another actor. The set of input ports I_a and output ports O_a of an actor a defines its interface.

Definition 2 *The connection relation C contains all pairs of output ports that are connected to input ports, i.e., $C = \bigcup_{a \in A} C_a$.*

For readability, for the rest of this thesis, $c = (o_a, i_b)$ denotes a connection between an output port of actor a and an input port of actor b. The type of a port c_{type} is determined by the type of the respective data tokens that are sent between the two ports.

A specific configuration of an embedded software is represented by the set of all actors A and their connection relations C:

Definition 3 *A software configuration is defined as a pair $S = (A, C)$, where A is the set of actors, and C is the connection relation.*

A reconfiguration transforms a software configuration $S = (A, C)$ to a new software configuration $S' = (A', C')$ by applying reconfiguration actions on the respective entities. Zhao & Li (2007b) supports four elementary reconfiguration actions, which are *startActor*, *stopActor*, *addConnector*, and *removeConnector*. Composite reconfiguration actions, such as *replaceActor* or *upgradeActor*, can be expressed via these four reconfiguration actions. However, they omit state transfer. As TimeAdapt supports also stateful actors, the decision was made to add the composite reconfiguration actions *upgradeActor* and *replaceActor* as explicitly supported reconfiguration actions that contain an inherent state transfer. In detail, TimeAdapt uses the following six reconfiguration actions:

- *addActor(a)*: Creates a new actor. This reconfiguration action is similar to Li's *startActor*. In case of a stateful actor, however, the action initialises the execution state with default values.

- *removeActor(a)*: Removes an actor. This reconfiguration action is similar to Li's *stopActor*. In case of a stateful actor, however, the action first removes execution state and then the respective actor is deleted.

- *replaceActor(a,a$'$)*: Replaces the implementation of the old actor with a new actor implementation. This action may lead to the requirement for an interface change. If the actor to be replaced contains state, this action executes a state transfer from the old actor to the new actor.

- *upgradeActor(a, a$'$)*: Replaces the implementation of the old actor with a new actor implementation. However, the interface of the old actor does not change, i.e., the new actor implements the same input and output port sets. Like the *replaceActor* reconfiguration action, the action is associated with state transfer between the old and the new actor.

- *connect(a, b, o_a, i_b)*: A connection is created between the output port o_a of actor a and the input port i_b of actor b. The type of the connection depends on the actual type of data being sent between the two ports.

- *disconnect(a, b, o_a, i_b)*: disconnects two actors that are connected via a connection.

3.2.2 Reconfiguration Model

The requirements, identified in Section 3.1, form the basis for the design of TimeAdapt, the time-adaptive reconfiguration model proposed in this thesis. This section focuses on two main aspects of the reconfiguration model: the guarantee that the software is in a consistent state at all times and the actual execution of a reconfiguration and reaction to incoming events. A consistent state requires transactional completeness and the maintenance of structural dependency relationships (Goudarzi, 1999). Transactional completeness is discussed in the next section, and structural dependency relationships are addressed, together with TimeAdapt's proposed scheduling mechanism, in Section 3.2.2.2.

3.2.2.1 Transactional Completeness

A transaction refers to the processing of a request from the time it is received by a software entity to the generation of a corresponding result by either the same software entity or through a chain of connected software entities (Li, 2009). Once a transaction is instantiated it should complete. This is referred to as "transactional completeness" (Zhang et al., 2005). Transactional completeness requires that an ongoing transaction is neither stopped nor destroyed by any ongoing reconfiguration. To achieve this, a synchronisation mechanism is needed to guarantee that any reconfiguration action, potentially interrupting an ongoing transaction, is executed only after the transaction completes. In literature this is referred to as quiescent or reconfiguration-safe state (Kramer & Magee, 1990).

As outlined in the review in Chapter 2, existing reconfiguration models for embedded software follow two orthogonal approaches, which were discussed in more detail in Section 2.1.3. They either bring the affected software entities into a reconfiguration-safe state before any modifications are executed and then execute modifications sequentially on the mutually-consistent software entities. Or they remove the need for such a state by applying a transitional approach in which the old and the new software configuration are executed in parallel. The transitional approach has some disadvantages, especially in terms of additional memory and processor costs caused by redundancy during reconfiguration (Li, 2009). Therefore, TimeAdapt drives affected software entities explicitly into a reconfiguration-safe state before sequentially executing the reconfiguration actions. This is realised by waiting until a currently active software entity, i.e., actor, has finished its internal computation. As the development of a synchronisation mechanism is outside the scope of this work, this thesis assumes that the underlying system model is equipped with suitable mechanisms, such as thread-counting (Soules et al., 2003) or reader-writer locks on functional interfaces (Wermelinger, 1997).

3.2.2.2 Scheduling Mechanisms

This section describes the scheduling mechanisms used by existing reconfiguration models to schedule an ongoing reconfiguration for execution, as well as the maintenance of dependency relations between software entities.

A scheduling mechanism determines how the centralised entity that executes reconfigurations reacts to events that occur during an ongoing reconfiguration. In this context, a reconfiguration can be seen as a task comprising n reconfiguration actions, or so-called jobs. In this section, the terms task and reconfiguration are used interchangeably, as well as the terms reconfiguration action and job. From the discussion of existing reconfiguration models, it emerged that they use either scheduling mechanisms that lead to a preemptive execution model, such as priority-based scheduling mechanisms, or mechanisms that lead to a transactional execution model, such as clock-driven scheduling (Liu, 2000). For reasons of completeness, the deadline-aware scheduling mechanism, which was not part of the discussed work, is added to this list of mechanisms considered for TimeAdapt. More specifically, the following mechanisms are discussed in more detail:

1. Priority-driven direct preemptive scheduling: Scheduling decisions are made whenever an incoming event arrives, as in Zhao & Li (2007b) (see Section 2.3.4)

2. Priority-driven delayed preemptive scheduling: Incoming events need to wait until an ongoing reconfiguration job is completed, as in Schneider et al. (2004) (see Section 2.3.3)

3. Clock-driven scheduling: A priori determination of task schedule as in Stewart & Khosla (1996) (see Section 2.3.6)

4. Deadline-Aware Scheduling: Controlled execution of current task under consideration of incoming events, e.g., as in Huang et al. (2009).

Priority-driven direct preemptive scheduling: An ongoing reconfiguration is directly pre-empted when an incoming event occurs, as the event will always have a higher priority. This scheduling mechanism is used for example in Li's reconfiguration model (see Section 2.3.4). The mechanism fulfils requirements R4 and R5 (the timely reaction to incoming events), although a number of issues arise.

Firstly, when there is a high event arrival rate, the direct pre-emptive approach cannot guarantee the completion of a reconfiguration, as the reconfiguration is constantly interrupted by high-priority events and because of that is subject to starvation. This would contradict requirement R3. Secondly, the direct pre-emption of a reconfiguration requires that reconfiguration actions themselves can be directly pre-empted (Zhao & Li, 2007b). However, time-consuming actions, such as state transfer actions, are not supported, contradicting requirement R7. Thirdly, a reconfiguration task can potentially be interrupted at any point in time, and might leave the system in an intermediary configuration with possible structural inconsistencies. Due to these issues, priority-driven direct preemptive scheduling was discarded as a candidate for TimeAdapt.

Priority-driven delayed preemptive scheduling: If an event occurs, an ongoing reconfiguration action is completed, before the event is processed. In this scheduling mode the incoming event might miss its deadline if the execution duration of the currently executing reconfiguration action is larger than the given deadline. Reconfiguration models that support the specification of a maximum-allowed reconfiguration execution duration may check whether an event processing deadline fits within this time before the next decision is made (Schneider, 2004). However, all events need to be known statically at system startup time and cannot change their deadline dynamically. This contradicts requirement R4. Also, the issue of starvation still remains if the event arrival rate is high as events are always associated with a higher priority and are directly processed if there is no ongoing reconfiguration action. Because of these issues, priority-driven delayed scheduling was discarded as a candidate for TimeAdapt. Note that if an event occurs, and there is no currently executing reconfiguration action, the priority-driven delayed preemptive scheduling mode directly processes the event.

Clock-driven scheduling: In a clock-driven scheduling mechanism, a schedule of all tasks is done before a system begins execution. This scheduling mechanism realises a transactional reconfiguration model, if the reconfiguration task is scheduled in a time slot that fits all reconfiguration actions. Typically, this type of scheduling requires that all parameters of all tasks and the tasks themselves are fixed and known (Liu, 2000).

An ongoing reconfiguration task can be interrupted in this scheduling mechanism only by events, such as other system tasks that are also known statically before system startup, e.g., the PBO reconfiguration model (Stewart & Khosla, 1996). This scheduling approach is inflexible as it requires the knowledge of all possible reconfiguration tasks and event arrival rates at system design time and does not support the reconfiguration of tasks that are known only at system runtime or events with varying event arrival rate and deadlines. This contradicts requirements R4 and R8. Due to this inflexibility, the clock-driven scheduling approach was discarded as a candidate for TimeAdapt.

Deadline-Aware Scheduling: Deadline-aware scheduling executes tasks in a controlled manner, while at the same time abiding by time constraints (Huang et al., 2009). The mechanism greedily schedules as many reconfiguration actions as possible, before an interrupting event is processed. However, the mechanism also tries to abide by an event's processing deadline.

This mechanism offers more flexibility than the other scheduling mechanisms, as it is dynamically applied on previously unknown reconfiguration sequences and it supports events that were unknown at system startup time (R4). The greedy scheduling of reconfiguration actions ensures that at least the currently active reconfiguration action is scheduled (R3), while processing incoming events in a timely manner (R5). As this scheduling mechanism addresses the key requirements for a time-adaptive reconfiguration model (R3-R5), we base TimeAdapt's scheduling mechanism on this approach. The next section discusses, how the key requirements of a time-adaptive reconfiguration model are met by TimeAdapt's deadline-aware scheduling mechanism.

3.2.2.3 Deadline-Aware Scheduling in TimeAdapt

This section investigates how the properties of a deadline-aware scheduling mechanism lead to a time-adaptive execution model, which tries to meet processing deadlines of previously unknown incoming events (R4, R5), while proceeding with an ongoing reconfiguration towards its completion (R3).

Meeting event deadlines: If an event occurs during an ongoing reconfiguration task, the scheduling mechanism needs to react dynamically to this event and its associated event processing deadline. On the one hand, completing the ongoing reconfiguration might not be feasible depending on how many actions there are still to execute and the size of the event deadline. The event deadline may be missed. On the other hand, a direct pre-emption of the reconfiguration may lead to problems already discussed in Section 3.2.2.2, such as possible starvation of a reconfiguration task and potential inconsistent configurations. Deadline-aware scheduling supports the controlled, partial execution of a reconfiguration task. Reconfiguration actions are scheduled in a greedy fashion as long as they can be executed within the event's processing deadline. This potentially results in an execution of only a sub-set of the remaining reconfiguration tasks. Depending on the time available for a given reconfiguration sequence, the outcomes of a reconfiguration might be the completion of sub-parts of the reconfiguration, or the completion of the overall reconfiguration. If the event processing deadline is at least as large as the time until completion of a currently executed reconfiguration action, this deadline can be met. Processing deadlines are not met when these deadlines are smaller than the execution duration of the currently executed reconfiguration action. As this work does not consider critical systems that have hard real-time requirements, a missed event deadline leads only to deterioration of system quality parameters, such as perceived signal quality, but not to the non-functioning of the system. A discussion of TimeAdapt's timing behaviour and its guarantees with respect to meeting processing deadlines is also done in Section 3.4.2.4. The next section discusses how the division of a reconfiguration into sub-sequences addresses the issue of ensuring reconfiguration completion.

Proceeding with an ongoing reconfiguration: Progress towards reconfiguration completion using deadline-aware scheduling depends on the time at which an incoming event occurs. If an incoming event occurs during an ongoing reconfiguration action, the deadline-aware scheduling mechanism prefers the completion of this action over the incoming event. The mechanism tries to greedily schedule as many reconfiguration actions as possible, before the event is processed. This has several implications on an ongoing reconfiguration. Firstly, the preference of an ongoing reconfiguration action over incoming events leads to the progress of the reconfiguration sequence towards its completion. Even when the processing deadline is so small that no further actions can be scheduled, the mechanism at least completes the current action. Secondly, the scheduling mechanism allows the potential execution of multiple reconfiguration actions, before an event is processed. This is in contrast to delayed preemptive approaches where events are given higher priorities and which only complete the current action before processing an event. The incremental scheduling of remaining reconfiguration actions leads to an eventual completion of the reconfiguration.

When multiple, subsequent events occur, an event is directly processed after its predecessor. As the scheduling mechanism always completes a reconfiguration action, there is no currently executing reconfiguration action, and the progress of a reconfiguration sequence depends on the available deadline. The scheduling mechanism schedules the next reconfiguration action for execution if its execution duration is within the processing deadline. In this case, progress towards a completion of the reconfiguration sequence is made. If the execution duration of the next reconfiguration action cannot be scheduled within the processing deadline, the event is directly processed. In this case, the potential threat of reconfiguration starvation remains. Section 3.4.2.4 provides an in-depth discussion of TimeAdapt's guarantees for reconfiguration completion.

In both cases, the partial execution of a reconfiguration sequence leads to the challenge of satisfying structural integrity requirements. This challenge is discussed in more detail in the next section.

Safe Software Configurations: Dependency relationships between software entities are potentially destroyed, if an ongoing reconfiguration is only partially executed. Therefore, it needs to be ensured that these relationships are maintained during a reconfiguration (Almeida et al., 2001). According to Zhang et al. (2005), a software configuration is defined as *safe*, if it maintains all dependency relationships. In their system model a software entity a is dependent on a software entity b, if there is a (unidirectional) communication between these components. This thesis transfers safe software configurations to TimeAdapt's reconfigurable dataflow system model. Software entities in this system model are connected via channels. The entity whose input port is connected via the channel is dependent on the other software entity, as data can only flow if the outgoing software entity computes some data and outputs it on the channel. A reconfiguration can temporarily disconnect the two software entities, however, before a computation takes place in the entities, it must be enforced that data can flow without loss through the channel. This leads to our definition of a safe software configuration:

Definition 4 *A safe configuration is defined as a configuration $S' = (A', C')$ for which is valid:* $\forall i \in I_b, b \in A'$ *then* $\exists\ o \in O_a, a \in A' : c = (o_a, i_b), c \in C'$

A safe software configuration in the reconfigurable dataflow system model is a configuration in which input ports are only temporarily blocked if there are no data tokens available, but never because the associated output port was removed. Locally, this means that each input port i in the resulting configuration $S' = (A', C')$ is either connected directly to a data source, generating input data, or connected to a respective output port o.

Figure 3.2 illustrates an example, comprised of four actors and connections between these actors. The reconfiguration to be scheduled and executed is illustrated to the right as well as possible interruption points of the reconfiguration due to incoming events. Figure 3.3 illustrates the resulting software configuration when the reconfiguration se-

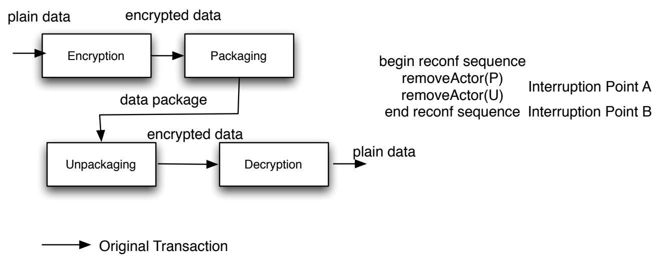


Fig. 3.2: Original actor configuration and the executed reconfiguration sequence

quence is executed until *point A*. In this case, the encryption actor and the decryption actor are currently disconnected. Here, the encryption actor would not be able to release the blocked unpackaging actor as a token can never arrive at the unpackaging actor's input port. Hence, the overall application is blocked and non-functioning.

Figure 3.4 illustrates the resulting software configuration, when the reconfiguration is interrupted at *point B*, i.e., after the execution of a removal reconfiguration action of the unpackaging actor (U). The resulting software configuration is safe, as all input ports of the decryption actor are connected.

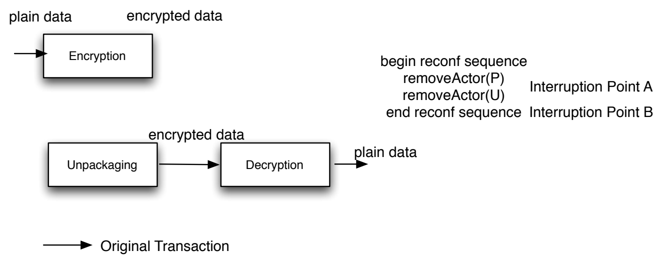


Fig. 3.3: Unsafe software configuration

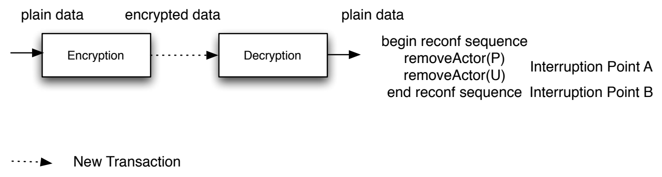


Fig. 3.4: Safe software configuration

Care must be taken that a reconfiguration always ends in a safe software configuration, whether it is fully completed or the configuration is an intermediary one where the reconfiguration is only partially executed. This is achieved in TimeAdapt by partitioning the reconfiguration into sub-sequences that transform a safe software configuration in a new safe software configuration. In the above example, the sub-sequence *removeP, removeU* denotes such a sub-sequence. These ordered sub-sequences are termed safe steps (Zhang et al., 2005).

Definition 5 *A step $p = \{r_1, r_2, \ldots, r_m\}$ is an ordered set and contains m reconfiguration actions, with $r_i \in R$.*

Definition 6 *A step p is said to be safe if it transitions from a software configuration s to a software configuration s', and both s and s' are safe.*

If a reconfiguration is performed as a sequence of safe steps and each safe step denotes a transition between two safe configurations, then during the reconfiguration the system is either in a safe software configuration or in a transition from a safe software configuration to another safe software configuration, i.e., currently executing a reconfiguration action of a safe step. If the software is in a safe software configuration, the dependency relationships between the software entities are maintained. As no reconfiguration action is executed, the software is functioning and no actor is blocked. If the software is in a transition from a safe software configuration, it is currently executing a reconfiguration action which is part of a safe step. Safe steps start and end in safe software configurations, so that no dependency relationships are destroyed, either before or after the execution of a safe step set. As safe steps are atomically executed, we can assume there are no intermediate configurations, and no dependency relationships destroyed.

As each reconfiguration safe step needs knowledge about the underlying software configuration and the current dependencies, the partitioning of a reconfiguration is orchestrated by the reconfiguration designer in the reconfiguration design phase. The partitioning algorithm is discussed in detail in Section 3.4.1

This section has given the rationale for the design of TimeAdapt's key element, its scheduling of ongoing reconfigurations based on deadline-aware scheduling. The next section discusses TimeAdapt's system assumptions as well as the formalisation of TimeAdapt's system elements. In the subsequent section the algorithms and processes that enable the model to meet its requirements for a time-adaptive reconfiguration model are discussed in detail.

3.3 TimeAdapt Reconfiguration Model

This section provides the set of system assumptions and notations used throughout the rest of this thesis. It also introduces the different system elements contained in TimeAdapt.

3.3.1 System Assumptions

TimeAdapt is executed on a single-processor embedded platform. The reconfiguration model requires the embedded software to be modelled according to the RDF system model as a set of actors A and their connection relationships. All actors have an unique numeric identifier that is used to address them in reconfiguration actions. Each actor provides reconfiguration capabilities to realise changes to their own input and output ports, such as setting and unsetting connections. Each actor has associated time estimation functions that are used to determine the estimated duration time of a specific action. These estimated times must be taken into account when determining how much time is available before an incoming event must be processed. The estimation functions and their respective values must be provided by the actor developer.

Reconfigurations, and their contained reconfiguration actions, are assumed to be already planned and available. It is further assumed that the reconfiguration trigger is known and formalised in the system, for example by using rules. A central entity, the reconfiguration manager, has access to all actors and executes modifications on the actors, such as loading and initialising new actors as well as stopping and removing old actors from the system. However, only a subset of actors $M = \{a_1, a_2, \ldots, a_m\} \subseteq A$ are subject to reconfiguration.

The reconfiguration manager runs in a single thread of execution, which may be interrupted by incoming events, such as events coming from the hardware. It is assumed that there is only a negligible delay from the time an event occurs until its occurrence is reported to the reconfiguration manager. Events are the only source of potential preemption of a reconfiguration in the system. An event does not need to be directly processed but can be postponed (Regehr, 2008). However, each event is associated with a desired maximum processing deadline, denoting a maximum desired time interval from the time the event was detected until its processing begins. The actual processing of an event is not reentrant, as the interleaved invocation of the same event processing code is disabled. However, in the time between the occurrence of an event and its actual processing, further events from the same or different event sources might occur.

3.3.2 Definition of Elements

A set of system elements can be derived from the previously discussed system assumptions that constitute the TimeAdapt reconfiguration model. The system elements are:

- A software configuration, which is defined in Definitions 2 and 3.
- A set of reconfigurable actors $M \subseteq A$.
- A set of reconfiguration actions $R = \{r_1, \ldots, r_n\}$, where $(n > 1)$. Each reconfiguration action r_i is an element from the set of supported reconfiguration actions, defined in Section 3.2.1. Each reconfiguration action r_i addresses either an existing actor $a \in M$ or a new actor and transforms the current system configuration into a new configuration. Each reconfiguration action has an associated time function t_e, which returns the estimated reconfiguration duration for the respective actor.
- A set of safe steps $P = \{p_1, \ldots, p_n\}$, where $(n \geq 1)$. Each $p_i \in P$ contains m reconfiguration actions, with $m >= 1$. A reconfiguration action is always contained in a single safe step, i.e., two safe steps do not contain the same reconfiguration action.
- A set of event sources $E = \{e_1, \ldots, e_n\}$. Each event source e_i emits events of a specific type with an associated event arrival rate λ_i. Each event has an associated event deadline t_d.
- A reconfiguration manager that has access to all actors of the software configuration. The reconfiguration manager is the entity that schedules and executes reconfiguration actions for execution.

- An event queue as part of the reconfiguration manager that stores incoming events for processing. The storing of events does not impose additional waiting time on the current system activities. Events in the queue are ordered according to their processing deadline.

3.4 TimeAdapt Processes and Algorithms

This section focuses on TimeAdapt's set of algorithms. As illustrated in Figure 3.5, TimeAdapt's algorithms are either applied at reconfiguration design time or reconfiguration runtime. One algorithm partitions reconfiguration actions into safe steps, at design time, while the remaining algorithms schedule safe steps according to event processing deadlines at runtime.

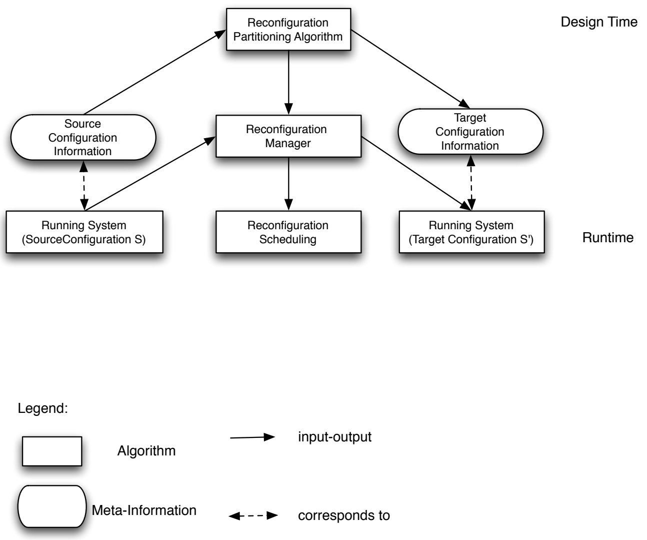

Reconfiguration Partitioning Algorithm
Design Time
Source Configuration Information
Reconfiguration Manager
Target Configuration Information
Running System (SourceConfiguration S)
Reconfiguration Scheduling
Running System (Target Configuration S')
Runtime
Legend:
Algorithm
input-output
Meta-Information
corresponds to

Fig. 3.5: TimeAdapt reconfiguration model overview

3.4.1 Reconfiguration Design Time

A reconfiguration developer specifies reconfiguration actions. Once a set of reconfiguration actions is defined, they must be grouped according to the dependency relationships of the addressed software entities. This results in a set of safe steps, each of which transforms a safe software configuration to a new safe software configuration. TimeAdapt includes a heuristic partitioning process that the reconfiguration designer can follow to map reconfiguration actions to safe steps. As input, this partitioning process requires a set of reconfiguration actions, as well as dependency descriptions between affected actors. It outputs a set of safe steps, representing atomic transformations from the current, safe source configuration to the desired target configuration (see Figure 3.6).

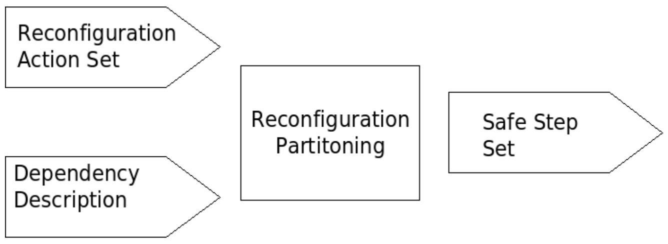


Fig. 3.6: Reconfiguration partitioning inputs and outputs

3.4.1.1 Reconfiguration Partitioning Heuristic

This section describes a heuristic for partitioning a set of reconfiguration actions into safe steps. As this process requires knowledge of the existing system configuration, it cannot be fully automated. However, a set of rules can be derived that reconfiguration developers can follow. This set of rules requires the explicit specification of dependency relationships between affected actors in the form of dependency descriptions.

Definition 7 *A dependency description Dep contains a set of actors that are dependent on a specific actor. An actor b is dependent on an actor a, if there is a channel between both actors, and this channel connects the input port of actor a to the output port of* *actor b, i.e., the relationship (o_a, i_b) is part of the connection relationship C of the current software configuration.*

The partitioning of reconfiguration actions into safe steps can be described as a mapping function F, with F being defined as $R \times Dep \rightarrow P$. R denotes the set of all reconfiguration actions, Dep denotes the set of all dependency descriptions, with $Dep = \{dep_1, dep_2, \ldots, dep_n\}$. It is assumed that a dependency description for the current configuration is given. P denotes the set of safe steps. F maps each reconfiguration action $r_i \in R$ to a specific safe step, using the dependency description $dep_i \in Dep$ of the affected actor to determine which actions need to be grouped together to reach a safe software configuration.

However, the actual partitioning also considers the type of reconfiguration actions. The reconfiguration action *upgradeActor* does not change the interface of an actor. As reconfiguration actions are either executed fully, or not at all, this reconfiguration action can be partitioned into its own safe step, without considering dependency relationships. Reconfiguration actions, such as *replaceActor*, can change the interface of an actor, as they add or remove input and output ports from the existing set. They need to be partitioned into the same safe step than the action that addresses the dependent actor. The reconfiguration actions *addActor* and *connectActor*, as well as the reconfiguration actions *removeActor* and *disconnectActor* are partitioned into the same safe step. If there are multiple *addActor* or *removeActor* actions, with dependent actors, they are also partitioned into the same safe step. In summary, the partitioning process obeys the following rules:

- An *upgradeActor* action can be partitioned into its own safe step.
- A safe step p_i that contains a $replaceActor(a)$ action, also contains the corresponding reconfiguration action $replaceActor(b)$, where b is dependent on a.
- An actor can only be removed, after all its connections are removed: A safe step

p_i that contains a $disconnect(a, b, o_a, i_b)$ action, also contains the corresponding reconfiguration action $removeActor(b)$.

- An actor can only be connected with other actors, after it has been created: A safe step set p_i that contains an $addActor(a)$ action also contains the corresponding reconfiguration action $connect(a, b, o_a, i_b)$.

Figure 3.7 illustrates the basic partitioning heuristic that reconfiguration designers can follow. If the reconfiguration action is an *upgradeActor* action, it is partitioned into its own safe step. For each other reconfiguration action, a new safe step must be created and the dependency descriptions of the respective actors must be retrieved. All reconfiguration actions that contain dependent actors must be grouped into the same set. This needs to be transitively repeated for all reconfiguration actions that are grouped into this set. The size of a safe step depends then on the dependency relationships of the underlying software entities.

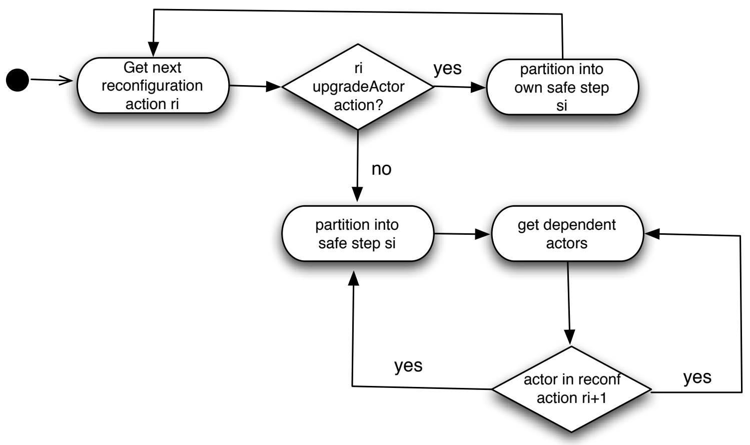

Get next reconfiguration action ri
ri upgradeActor action?
yes
partition into own safe step si
no
partition into safe step si
get dependent actors
yes
actor in reconf action ri+1
yes

Fig. 3.7: Partitioning algorithm

3.4.1.2 Partitioning Example

Figure 3.8 and 3.9 illustrate two examples for the partitioning of two reconfiguration sequences on an example actor network that consists of six actors. The example was taken from (Cheong, 2003) and illustrates the beaconless communication protocol software. The actors *InitiateRouterMessage* and *ProcessMessage* receive input data from either the application or an initialisation process. The actor *SendMessage* is connected to these two actors via two channels. This actor would block, if there is no data available on the channels, i.e., if the connected actors would not produce any data. Hence, the actor is dependent on its two input actors. The same is valid for the *GenericComm* actor that is dependent on the *SendMessage* actor.

Figure 3.8 illustrates the partitioning result for interface-changing *replaceActor* reconfiguration actions. In this example, the final number of reconfiguration actions in a single safe step is three. The reconfiguration action *replaceActor(ProcessMessage)* is first partitioned. The *ProcessMessage* actor has a single dependent actor, the *SendMessage* actor. The partitioning process then searches in the reconfiguration actions for a reconfiguration action that addresses this actor. Hence, the *replaceActor(SendMessage)* is partitioned next. The *SendMessage* actor has as dependent actor the *GenericComm* actor. However, there is no reconfiguration action that addresses this actor. The last remaining reconfiguration action is the *replaceActor(InitialRouterMessage)*, which will be partitioned into the same safe step.

Figure 3.9 illustrates the partitioning result for the non-interface changing *upgradeActor(ProcessMessage)*, as well as for the interface-changing reconfiguration actions *replaceActor(SendMessage)* and *replaceActor(GenericComm)*. The non-interface changing reconfiguration action *upgradeActor(ProcessMessage)* is partitioned into its own safe step. In case of an interruption of the reconfiguration process, the interface of the actor has not changed and data can still be transferred between the actors. However, the two reconfiguration actions *replaceActor(SendMessage)* and *replaceActor(GenericComm)* are partitioned into the same safe step, due to the dependency relationship of their respective actors.

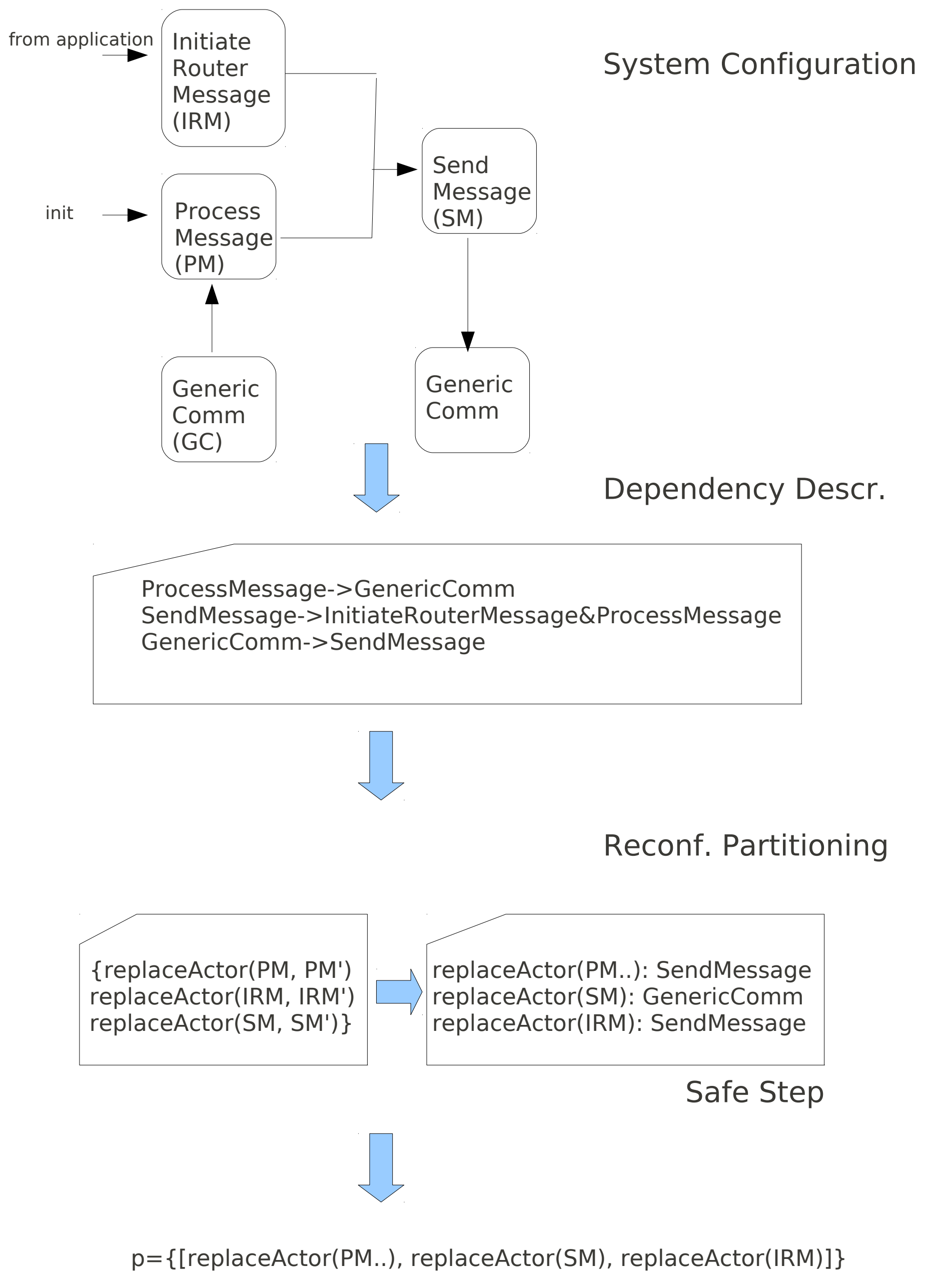

from application
Initiate Router Message (IRM)
System Configuration
init
Process Message (PM)
Send Message (SM)
Generic Comm (GC)
Generic Comm
Dependency Descr.
ProcessMessage->GenericComm
SendMessage->InitiateRouterMessage&ProcessMessage
GenericComm->SendMessage
Reconf. Partitioning
{replaceActor(PM, PM')
replaceActor(IRM, IRM')
replaceActor(SM, SM')}
replaceActor(PM..): SendMessage
replaceActor(SM): GenericComm
replaceActor(IRM): SendMessage
Safe Step
p={[replaceActor(PM..), replaceActor(SM), replaceActor(IRM)]}

Fig. 3.8: Partitioning example: Interface-changing actions

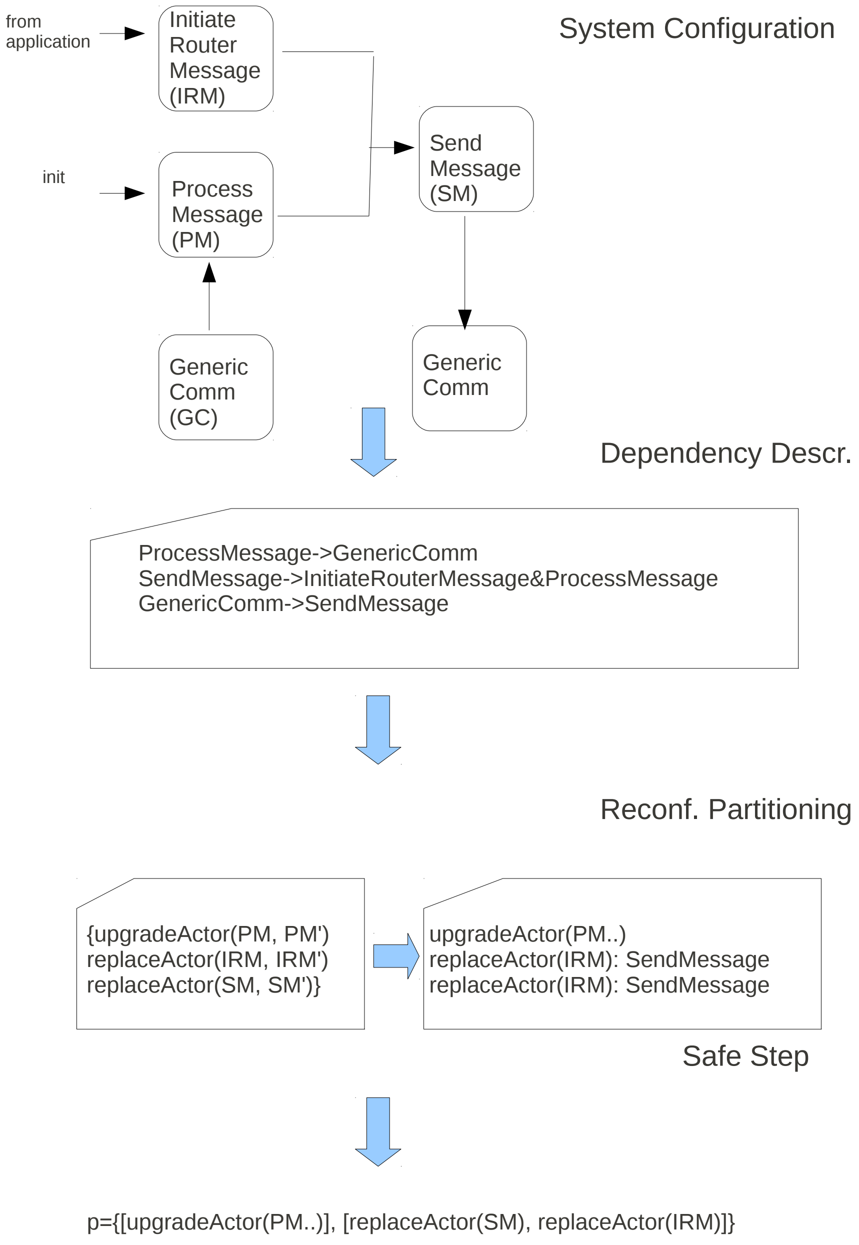


Fig. 3.9: Partitioning example: Non-interface changing and interface-changing actions

3.4.2 Reconfiguration Runtime

This section describes the algorithms for the initialisation and scheduling of reconfigurations. Reconfiguration initialisation is executed when a reconfiguration is started, whereas reconfiguration scheduling is executed during the actual execution of a reconfiguration. In contrast to the algorithms in the reconfiguration design phase, these algorithms are executed without additional user input, and are automated.

3.4.2.1 Reconfiguration Execution Initialisation

As a safe step is atomically executed, the order of the reconfiguration actions within a safe step is of no relevance. However, for this thesis we address a single-processor embedded platform, and at each point in time only a single reconfiguration action can be executed. Therefore, reconfiguration actions and safe steps are assumed to be totally ordered.

Existing work has defined partial temporal orders for related reconfiguration actions, i.e., actions that address the same actor (Wermelinger, 1997). As two safe steps can contain reconfiguration actions that address different actors, this partial temporal order $<$ needs to be extended to span across different safe steps. If $p_i < p_j$, with $p_i, p_j \in P$, then all reconfiguration actions contained in p_i are executed before reconfiguration actions contained in p_j. This temporal order must include the following relationships (Wermelinger, 1997):

1. A safe step p_i that contains a reconfiguration action r_i, must be executed before a safe step p_j containing reconfiguration action r_j, if action r_j executes on an actor that is dependent on the actor addressed by reconfiguration action r_i.

2. A safe step p_i first executes reconfiguration action r_i , and then all its dependent reconfiguration actions r_j.

These rules require the dependency descriptions *Dep* for each affected actor, which are also used in the reconfiguration design phase.

Initialisation Example Figure 3.10 illustrates an initialisation of a reconfiguration sequence, comprising three safe steps. Safe step p_b contains the reconfiguration action r_b (*upgradeActor(b)*), which addresses actor b. Actor b is dependent on actor a and hence the resulting partial ordering results in $p_a < p_b$. The same goes for safe step sets p_b and p_c, resulting in the final total ordering of $p_a < p_b < p_c$.

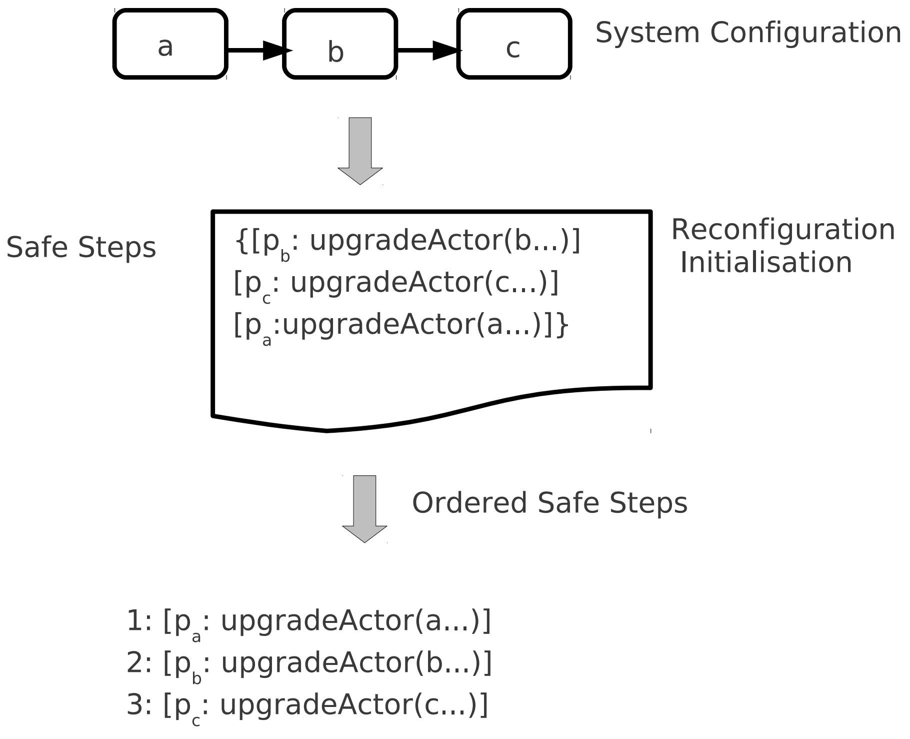

a
b
c
System Configuration
Safe Steps
{[p_b: upgradeActor(b...)]
[p_c: upgradeActor(c...)]
[p_a:upgradeActor(a...)]}
Reconfiguration Initialisation
Ordered Safe Steps
1: [p_a: upgradeActor(a...)]
2: [p_b: upgradeActor(b...)]
3: [p_c: upgradeActor(c...)]

Fig. 3.10: Initialisation example

3.4.2.2 Reconfiguration Execution Scheduling

After the reconfiguration is initialised and the input safe steps are totally ordered, the reconfiguration manager starts the actual execution of the reconfiguration actions, contained in the safe steps. The reconfiguration manager has access to all actors in the system and is responsible for executing the reconfiguration actions on the respective actor. Execution of a reconfiguration action is *atomic* and *isolated*. Atomic execution implies that once the reconfiguration action starts its execution on a respective actor, it is either run to completion or its effects do not take place (David & Ledoux, 2006b). Isolation implies that the action's execution is performed without interleaving with other operations (Zhang et al., 2005). Both properties must be supported by the underlying system model, for example by providing synchronisation mechanisms, as discussed in Section 3.2.2.1.

A reconfiguration execution then first brings actors to be reconfigured into a reconfiguration safe state. Given that all these actors are in such a state, the reconfiguration actions themselves are executed, transforming the system from the source configuration S to the desired target configuration S', if no event occurs during the ongoing reconfiguration.

If an event occurs, the reconfiguration manager is directly notified and uses the deadline-aware scheduling algorithm with the event's associated processing deadline as a parameter. TimeAdapt provides two scheduling algorithms, so-called modes, that differ in the granularity of scheduled actions. For the following discussion we assume that only a single incoming event occurs. The generalisation to multiple events is discussed in Section 3.4.2.3.

Pessimistic Scheduling Mode The basic scheduling algorithm is described in Algorithm 1, with a set of safe steps $P = \{p_1, p_2, \ldots, p_n\}$. Note that r_{ij} denotes the $i-th$ reconfiguration action in safe step p_j, t_d denotes an event's processing deadline, $t_e(r_{ij})$ the estimated execution duration of reconfiguration action r_{ij}, t_e denotes the total estimated execution duration of the safe step p_j, t_t denotes the (actual) total execution duration of the safe step p_j and all its predecessors, and $t_t(r_{ij})$ denotes the (actual) total execution duration of reconfiguration action r_{ij} (see Table 3.1). Due to its conservative scheduling this algorithm is also called pessimistic mode.

The pessimistic scheduling mode differs two cases, depending on whether a safe step

```
1  $t_t \leftarrow 0$ // On reception of incoming event do
2  if $p_j$ active then
       // complete all actions in safe step
3      foreach $r_{ij} \in p_j$ do
4          execute $r_{ij}$
           // update total execution duration time
5          $t_t \leftarrow t_t + t_t(r_{ij})$
6      end
7  end
   // try to schedule safe steps $p_{j+1}, \ldots, p_n$, if $t_t < t_d$
8  // As long as there are reconfiguration actions
9  while $P \neq \emptyset$ and $t_t < t_d$ do
       // get estimated total duration time of actions that are in next
          safe step $p_r$
10     foreach $r_i \in p_r$ do
11         $t_e \leftarrow t_e + t_e(r_i)$
12     end
13     if $t_e < t_d$ then
14         foreach $r_i \in p_r$ do
15             execute $r_i$ on respective actor atomically;
16             $t_t \leftarrow t_t + t_t(r_i)$
17         end
           // update remaining available time
18         $t_d \leftarrow t_d - t_t$  $t_e \leftarrow 0$
19     end
20     else
           // stop scheduling safe steps
21         return;
22     end
23 end
```

Algorithm 1: Pessimistic scheduling mode of safe steps

Symbol	Function
p_j	current active safe step
r_{ij}	i-th reconfiguration action in p_j
t_d	event processing deadline
t_e	estimated total execution duration of safe step
$t_e(r_{ij})$	estimated execution duration of r_{ij}
t_t	actual total execution duration of safe step p_j and predecessors
$t_t(r_{ij})$	actual execution duration of r_{ij}

Table 3.1: Overview of pessimistic mode parameters

is currently executing or not, when an event occurs. A safe step is executed, if either a reconfiguration action that is part of a safe step is currently executing, or there are still remaining reconfiguration actions in this safe step that need to be executed. If an event occurs while a safe step is executing, this safe step needs to complete first so that a safe software configuration is reached (Lines 2-6). The mechanism then checks whether it can still schedule subsequent safe steps (Line 9). If its estimated execution duration t_e of the next, subsequent, safe step falls within the deadline, this safe step is atomically executed (Line 10-19). Otherwise, the mechanism directly pre-empts a reconfiguration sequence. If an event occurs while no safe step is executing, the pessimistic scheduling mode executes Lines 7-19, i.e., it tries to schedule as many safe steps as possible within the processing deadline.

Figure 3.11 illustrates the results that are possible when using the pessimistic scheduling mode by means of a reconfiguration example, comprised of two safe steps $p1$ and $p2$. $p1$ contains two reconfiguration actions and $p2$ contains three reconfiguration actions. The end result depends on the point in time when an incoming event occurs, illustrated by an arrow in Figure 3.11, as well as the incoming event's processing deadline t_d and the estimated time duration t_e of safe steps:

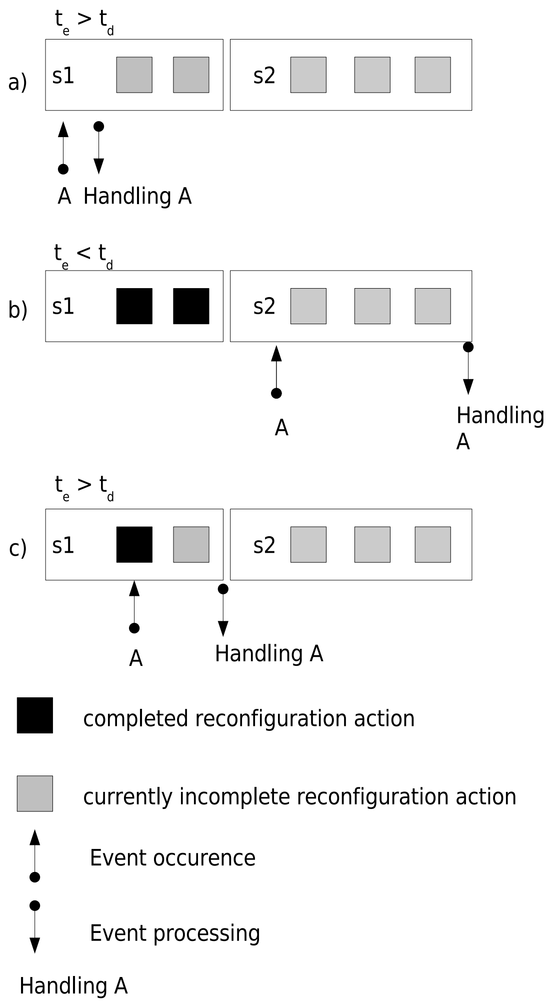


Fig. 3.11: Different outcomes for the pessimistic scheduling mode

a) An incoming event occurs when no safe step is currently executing and the estimated duration of this safe step is larger than the given deadline. In this case, the reconfiguration is directly terminated as the current software configuration is safe and a direct context switch to the processing of the event can be executed. This results in no change in the given configuration before the event is processed.

b) An incoming event occurs when no safe step is currently executing and the estimated duration of this safe step is smaller than the given deadline. In this case, all actions in the safe step are executed atomically. If there is still some time left after the reconfiguration duration, the algorithm tries to greedily schedule the next safe step. The result is either an intermediate or the target configuration. For example, in Figure 3.11 b), the incoming event occurs before safe step $p2$ is executed. In this example, the atomic execution of the safe step results in the target configuration.

c) An incoming event occurs during the execution of a safe step. In this case, the currently active safe step finishes execution. Depending on the remaining time, the next safe step set is either scheduled (see b) or a direct context switch to the incoming event is executed (see a). In this case, the processing deadline of an event might be missed, if the execution duration of the currently active safe step exceeds this deadline.

The pessimistic scheduling mode ensures that the software is transformed from a safe system configuration to a new safe system configuration, given that the estimated time of this safe step is large enough. However, care must be taken that the estimated time for each reconfiguration action is at least as large as its real execution duration. As the pessimistic mode does not support the revoke of reconfiguration actions, the estimated execution duration of a safe step needs to be its worst-case execution duration to ensure the adherence of time constraints.

Optimistic Scheduling Mode Introducing the possibility to revoke reconfiguration actions within a safe step means that a scheduling mechanism could decide during the execution of a safe step whether to continue that step or to revoke a reconfiguration action. Analogous to optimistic transaction concurrency control in data-base systems (Coulouris et al., 2005), we call this algorithm optimistic. The optimistic scheduling algorithm does not consider safe steps, but single reconfiguration actions. The implications of this are illustrated in Figures 3.12 and 3.13. Figure 3.12 illustrates the original system configuration, already introduced in Section 3.4.1.2. The reconfiguration to be scheduled and executed is illustrated to the right. Figure 3.13 illustrates the resulting system configuration when the reconfiguration sequence is executed using the optimistic mode until a certain so-called interruption point. In this example, a single *replaceActor* action is executed, until control is handed to the processing of the interrupting event. However, the *replaceActor* action can potentially change the interface of the actor, such as changing the data type of the output port. In this case, the resulting system configuration is unsafe, as the connection between the *ProcessMessage* actor and the *SendMessage* actor is interrupted.

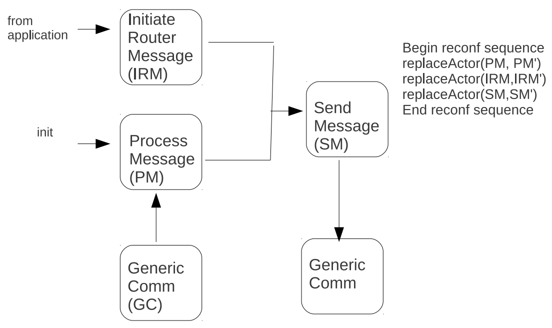


Fig. 3.12: System configuration before optimistic scheduling mode executes

To avoid unsafe system configurations that results because of the non-atomic execution of safe steps, additional constraints and assumptions need to be introduced for this optimistic mode. The biggest constraint and difference to the pessimistic mode is

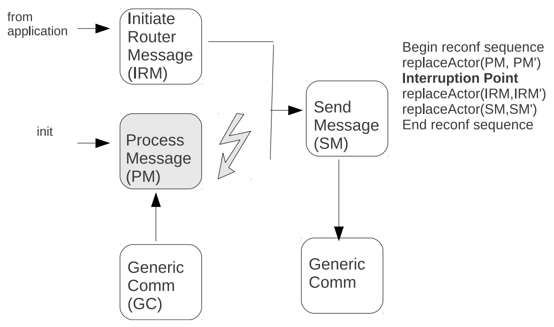


Fig. 3.13: System configuration after execution of a single reconfiguration action

that interface-changing actions, such as *replaceActor* actions are not supported. This constrains this mode to use reconfiguration actions, such as *upgradeActor*, *addActor*, and *removeActor*. However, additional constraints on *addActor* and *removeActor* actions need to be introduced. Whereas in the pessimistic mode newly created actors are activated directly at startup time, in the optimistic mode new actors are only activated, when the complete reconfiguration has completed. The same is valid for the deletion of actors. Actors are only deleted, once the overall reconfiguration has completed.

One main assumption that is introduced for the optimistic mode is that all supported reconfiguration action types have an associated undo-action. TimeAdapt's optimistic scheduling mode exploits the possible division of an *upgradeActor* reconfiguration action into two different phases to ease the potential revoke of this action, see Figure 3.14. The first phase transfers the state of a stateful actor. In the second phase, connections of dependent actors that are connected to the old actor are updated to connect to the new actor. A revoke in the state-transfer phase results in the direct interruption of the phase, whereas a revoke in the upgrade-connection phase results in the re-connection of dependent actors to the old actor. *AddActor* and *removeActor* reconfiguration actions can be seen as *upgradeActor* actions with an empty actor, and can therefore also be divided into these phases. The state-transfer phase of an *addActor* action creates the new actor, and the update-connection phase connects the input ports of the newly created actor to existing actors in the configuration. The state-transfer phase of the *removeActor* action is an empty phase, and the update-connection phase disconnects dependent actors from this actor.

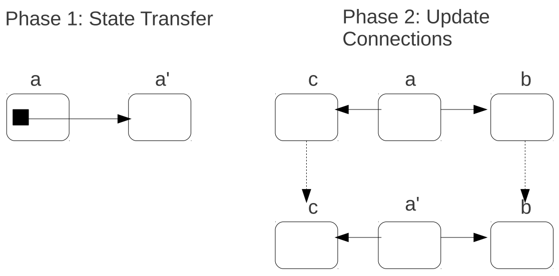


Fig. 3.14: Reconfiguration action phases

Instead of an estimation function that returns the estimated duration of the overall reconfiguration action, the optimistic mode requires two estimation functions: one to estimate the time to update dependent connections t_{uc} , and one to estimate the time it takes to revoke dependent connections t_r.

We describe the optimistic scheduling algorithm in Algorithms 2 and 3. Algorithm 2 illustrates the steps taken when the current reconfiguration action is in the state transfer phase, whereas Algorithm 3 illustrates the steps taken when the current reconfiguration action is in the upgrade connection phase. Table 3.2 shows the parameters used for the optimistic scheduling mode.

If an event occurs during the state transfer phase, the optimistic scheduling mode completes this phase first and then decides whether to process the event (Lines 14-15), or to continue the reconfiguration action by updating the connections (Lines 6-13). In the state transfer phase the state of a stateful actor is transferred from the old actor a to the new actor a'. If the estimated time to update connections is smaller than the processing deadline, the *update_ connection* phase is executed. Otherwise, the reconfiguration is interrupted and a direct context switch to the incoming event is executed. In this case, the old actor remains active. When the reconfiguration is scheduled for execution again,

```
// As long as there are reconfiguration actions
while $R \neq \emptyset$ do
    // On reception of incoming event do
    $cP \leftarrow$ current phase of reconfiguration action
    // state transfer phase
    if (cP = "state transfer") then
        complete state transfer phase
        $t_t \leftarrow$ time to finish state transfer phase
        // estimate time to update connections
        if ($t_{uc} + t_t < t_d$) then
            foreach $b \in A$ do
                // actors $a$ and $b$ are connected
                if $(o_a, i_b) \in C$ then
                    // udpate input port $i_b$ of actor $b$ to connect to
                       output port $o'_a$ of new actor $a'$
                    update connections($a', b$)
                    // remove reconfiguration action from reconfiguration
                       sequence
                    remove $r_i$
                end
            end
        end
        else
            // process incoming event
            return;
        end
    end
    // see Algorithm 3

end
```
Algorithm 2: Optimistic scheduling mode of safe steps: State transfer phase

```
   // update connection phase
20 else
       // determine whether to revoke or to complete action
21     if $t_{uc} < t_d$ then
           // continue update connection phase
22         foreach $b \in A$ do
               // actors $a$ and $b$ are connected
23             if $(o_a, i_b) \in C$ then
                   // udpate input port $i_b$ of actor $b$ to connect to output
                      port $o'_a$ of new actor $a'$
24                 update connections($a', b$)
25                 $t_t \leftarrow t_t$ + time to update connection
26             end
27         end
28         remove $r_i$
29     end
30     else
           // revoke current action
31         foreach $b \in A$ do
               // actors $a$ and $b$ are connected
32             if $(o_a, i_b) \in C$ then
                   // udpate input port $i_b$ of actor $b$ to connect to output
                      port $o_a$ of old actor $a$
33                 update connections($a, b$)
34                 $t_t \leftarrow t_t$ + time to revoke connection
35             end
36         end
37         remove $r_i$
38     end
39 end
```

Algorithm 3: Optimistic scheduling mode of safe steps: Update connection phase

Symbol	Function
p_i	current active safe step
r_i	i-th reconfiguration action in p
t_d	event processing deadline
a	old actor
a'	new actor
t_{uc}	estimated update connection time
t_r	estimated time to revoke connections

Table 3.2: Overview of optimistic mode parameters

the state transfer phase starts with the transfer of the actor's most current state (see Algorithm 2).

If an incoming event occurs during the update-connection phase, two outcomes are possible. If the estimated time to update connections t_{uc} is smaller than the processing deadline, the currently executing *update_connection* phase finishes execution (Lines 20-30). Otherwise, a revoke of the reconfiguration action is executed by reconnecting the input ports of dependent actors, which were previously connected to the newly replaced actor a', back to the old actor a (Lines 31 -40).

The end result of the optimistic mode depends on the size of the event processing deadline and the current phase of the reconfiguration action (see Figure 3.15):

a) A state transfer phase is currently executing and the estimated time to update connections is smaller than the event processing deadline. In this case, after the state transfer phase has been completed, the update-connection phase is executed, see case c) or d)

b) A state transfer phase is currently executing and the estimated time to update connections is larger than the event processing deadline. In this case, after the

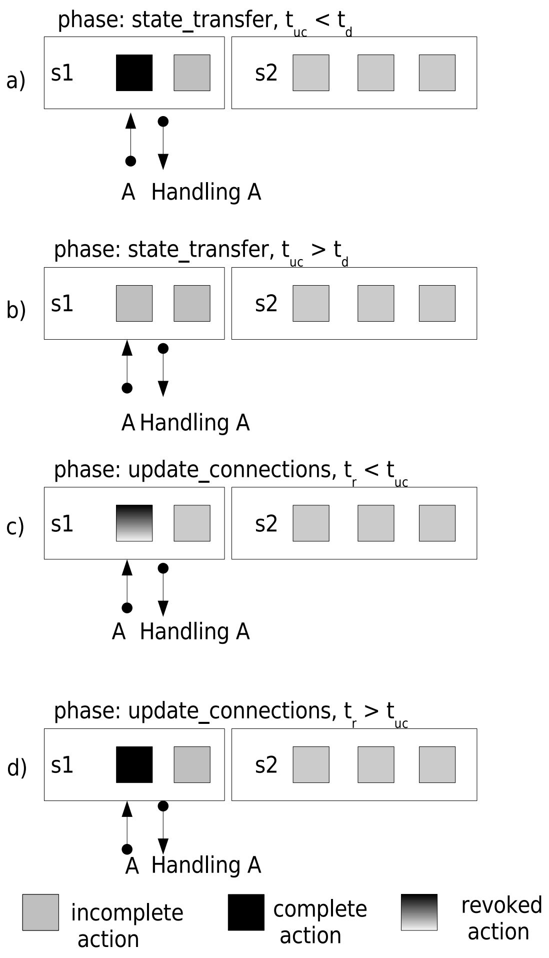

phase: state_transfer, $t_{uc} < t_d$
a)
s1
s2
A
Handling A
phase: state_transfer, $t_{uc} > t_d$
b)
s1
s2
A
Handling A
phase: update_connections, $t_r < t_{uc}$
c)
s1
s2
A
Handling A
phase: update_connections, $t_r > t_{uc}$
d)
s1
s2
A
Handling A
incomplete action
complete action
revoked action

Fig. 3.15: Different outcomes for the optimistic scheduling mode

state transfer phase has been completed, the ongoing reconfiguration is paused and the incoming event is processed.

c) An update-connection phase is currently executing and the estimated time to complete this action is larger than the processing deadline. In this case, actors, which were already connected to the new actor a', are connected back to the old actor a. After the revoke-phase has finished execution, the event is processed. Note that in this case the processing deadline might be exceeded if the revoke-phase also has a larger execution duration than the processing deadline.

d) An update-connection phase is currently executing and the estimated time to complete this action is smaller than the processing deadline. In this case, the connection-phase is completed by updating all actors whose input ports are connected to output ports of the new actor. If there are reconfiguration actions and execution duration left, the state transfer phase of the next reconfiguration action is executed.

Discussion Whether to use the pessimistic or optimistic mode depends on the parameter settings of the embedded software itself, such as the size of deadlines, and reconfiguration action execution durations. There is a tradeoff between generality and timeliness, which needs to be considered by the application developer.

The pessimistic mode can be applied on all reconfiguration action types discussed in Section 3.2.1. However, the timeliness of processing an incoming event depends on the size of this deadline, see Section 3.4.2.4 for a more detailed discussion on TimeAdapt's timeliness guarantees. The pessimistic mode is of advantage, when the processing deadlines are in general higher than the reconfiguration action execution durations. In this case, the pessimistic mode can schedule additional reconfiguration actions, without missing an event's deadline. Also, the pessimistic mode in general has a lower scheduling overhead compared to the optimistic mode.

The optimistic mode does not support interface-changing *replaceActor* actions, and the remaining actions are divided into a state-transfer and an update-connection phase. This more fine-grained scheduling mechanism allows a dynamic decision as to whether to continue or abort a currently executing reconfiguration action. This is especially of advantage, when deadlines are much smaller than reconfiguration action execution durations. In this case, deadlines can be met by the optimistic scheduling mode, in contrast to the pessimistic mode.

3.4.2.3 Handling of Multiple Events

The scheduling modes were discussed with respect to a single incoming event. However, embedded software is subject to multiple events, either from the underlying system or from the environment. As TimeAdapt can only process a single event at each point in time, these events may be potentially emitted at the same time and are queued by the reconfiguration manager according to their associated deadline. TimeAdapt then processes events subsequently. If an event's processing deadline is very small, TimeAdapt returns directly from scheduling further reconfiguration actions and processes the event. If an event's processing deadline allows the additional execution of reconfiguration actions, TimeAdapt allows the faster completion of an ongoing reconfiguration.

3.4.2.4 TimeAdapt Guarantees

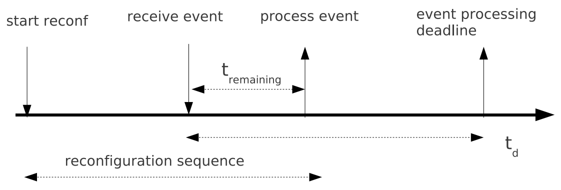
This section first discusses the conditions under which TimeAdapt can guarantee the meeting of an event's processing deadline t_d, and under which conditions TimeAdapt fails to meet an incoming event's deadline. It then discusses when a reconfiguration sequence is completed, and in which scenarios reconfiguration completion cannot be guaranteed.

Meeting Event Deadline TimeAdapt meets an event's processing deadline t_d in the pessimistic mode if t_d is at least as large as the remaining execution duration of the currently active safe step p_i. TimeAdapt meets an event's processing deadline t_d in the optimistic mode if t_d is at least as large as the remaining execution duration of the currently executing reconfiguration action.

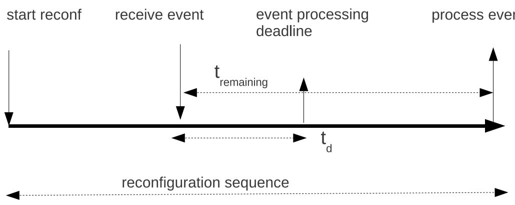
Figure 3.16 illustrates the timeline of a reconfiguration execution and the various times that are taken into account. To better compare the timing behaviour of the pessimistic and optimistic mode, we consider a safe step to be comprised of a single reconfiguration action. In the pessimistic mode $t_{remaining}$ denotes the time to finish a safe step, whereas in the optimistic mode $t_{remaining}$ denotes the time to either complete a phase or revoke a phase.

Figure 3.16(a) illustrates the case, when an incoming event's deadline is larger than $t_{remaining}$. In this case, the deadline is met by both modes of TimeAdapt.

Figure 3.16(b) illustrates the case, when an incoming event's deadline is smaller than $t_{remaining}$. In this case, the remaining time to complete a safe step in the pessimistic mode, or the completion, or revoke time of a currently active reconfiguration action phase is larger than the processing deadline t_d, and as a result TimeAdapt will miss this deadline.



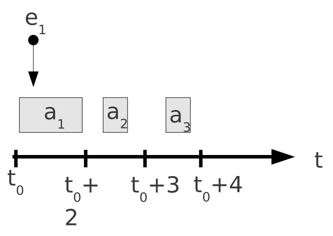
(a) Meeting deadlines



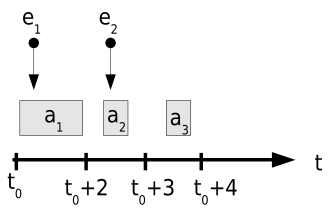
(b) Missing deadlines

Fig. 3.16: Timing behaviour of TimeAdapt

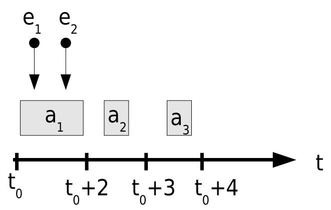
Progress towards Reconfiguration Completion TimeAdapt makes progress with an ongoing reconfiguration, if it can execute reconfiguration actions within the processing deadline. The progress of a reconfiguration depends on when an incoming event occurs, and the scheduling mode used. Figure 3.17 illustrates the different possible cases by means of a reconfiguration sequence comprised of three reconfiguration actions a_1, a_2, a_3. a_1 takes 2 time units to execute, whereas a_2, and a_3 take 1 time unit to execute.



(a) Single incoming event



(b) Two incoming events on different actions



(c) Two incoming events on same action

Fig. 3.17: Point in time of incoming events

Figure 3.17(a) illustrates the case, when a single incoming event occurs during an ongoing reconfiguration sequence. The pessimistic mode completes at least the currently executing action a_1, and depending on the event's deadline can schedule additionally the reconfiguration actions a_2 and a_3. In this case, the pessimistic mode always makes progress with an ongoing reconfiguration, as it favors an executing reconfiguration action over the direct processing of an incoming event. The optimistic mode, however, cannot guarantee the completion of the executing reconfiguration action a_1, as it depends on the type of phase that is active, when the incoming event occurs, and the size of the deadline. If the incoming event occurs during the state-transfer phase, the optimistic mode only makes progress with the reconfiguration if the incoming event's associated deadline is large enough to complete the reconfiguration action. The same holds for the update-connection phase, as this phase is only completed if it fits within the deadline.

Figures 3.17(b) and 3.17(c) illustrate the case where multiple, subsequent events occur during an ongoing reconfiguration. Note that these events all have the same associated deadline, and are processed in the order of their occurrence. In Figure 3.17(b) the events occur on different reconfiguration actions. Event e_1 occurs when reconfiguration action a_1 is executing, whereas event e_2 occurs when reconfiguration action a_2 is executing. It is further assumed that the processing deadline t_d of incoming event e_1 is too small to schedule additionally reconfiguration action a_2. With these assumptions, the scenario can be mapped to the already discussed case, illustrated in Figure 3.17(a). The pessimistic mode makes progress with an ongoing reconfiguration, independent of the associated deadline, as it at least completes action a_1 to process event e_1, and action a_2 to process event e_2. The optimistic mode makes progress with the reconfiguration only, if the associated deadline is at least large enough to complete the current reconfiguration action.

Figure 3.17(c) illustrates the case, when multiple, subsequent events occur on the same reconfiguration action. In this case, the progress of a reconfiguration action when dealing with event e_2 depends on its associated deadline. As event e_2 occurs during TimeAdapt's scheduling for its predecessor e_1, it is queued in the event queue. Depending on its remaining deadline, the pessimistic mode can either decide to schedule the next action, such as a_2, or aborts the reconfiguration. However, in cases when the deadline is smaller than the estimated execution duration of the next reconfiguration action, there is no progress with an ongoing reconfiguration. The same holds for the optimistic mode, that decides, depending on the current phase of the last executed reconfiguration action. In conclusion, TimeAdapt's pessimistic and optimistic mode guarantees reconfiguration progress, if event deadlines are at least as large as the remaining reconfiguration action execution durations.

3.5 Summary

In this chapter, we described the design of TimeAdapt. TimeAdapt is defined on a suitable system model that defines the underlying system entities. The reconfiguration model itself realises modifications on this system model. We presented the rationale for TimeAdapt's underlying system model and a detailed description of TimeAdapt itself. The combination of all mechanisms and algorithms are designed to address the full set of requirements of a time-adaptive reconfiguration model. See also Table 3.3 for a summary. Requirements that are only partially fulfilled by TimeAdapt are denoted in parentheses.

TimeAdapt is designed for software modelled according to the reconfigurable dataflow system model (RDF). RDF is a well-defined, abstract system model that can represent embedded software with a dataflow-based computational model, in which entities send data in a non-blocking manner and read data in a blocking manner. This system model was chosen as it represents a wide range of embedded system software, from signal processing to stream-oriented systems. Its abstract definition allows multiple implementations that can target a variety of system platforms, from single, centralised sensor nodes to more complex embedded systems whose software is potentially distributed. Therefore, this system model fulfils requirement R1. Also, the system model supports stateless and stateful system entities and hence does not impose any constraints, fulfilling requirement R7.

TimeAdapt itself uses a deadline-aware scheduling mechanism that results in a partial reconfiguration execution, depending on the event processing deadline. The partial execution of a reconfiguration means that event processing deadlines can be met, since the overall reconfiguration does not need to be completed, fulfilling requirement R4 and R5 partially. The scheduling mechanism used is oblivious to the size of a reconfiguration and supports any reconfiguration sequence length (requirement R6). Requirement R8 is partially fulfilled, as the mechanism requires time estimation functions for reconfiguration actions, and requires knowledge about the system configuration to determine safe steps.

Dependency relationships between actors determine sub-sequences, or so called safe steps, that ensure their maintenance even in the presence of partial reconfigurations. These safe steps are determined by the reconfiguration designer at reconfiguration design time and represent input for the reconfiguration manager, which executes the safe steps at reconfiguration execution duration. Along with a pre-requisite that TimeAdapt leverages an existing synchronisation mechanism, such as reader-writer locks, the partitioning into safe steps fulfils requirement R2, namely the guarantee of a correct system before, and after a reconfiguration.

A scheduling algorithm is executed when an incoming event occurs during an ongoing reconfiguration. TimeAdapt contains two scheduling algorithms that are part of the reconfiguration manager. Both algorithms realise the deadline-aware scheduling mechanism, as they schedule reconfiguration actions in a greedy fashion, while at the same time trying to make progress with an ongoing reconfiguration. The algorithms differ in the level of granularity applied. The incremental execution of reconfiguration actions leads to the eventual completion of a reconfiguration, fulfilling requirement R3.

The next chapter describes TimeAdapt's implementation on mappings of the abstract system model to components of our component model. Chapter 5 describes the evaluation of the proposed reconfiguration model on a real embedded platform and compares its results to implementations of a transactional and a preemptive reconfiguration model.

Req #	Requirements	TimeAdapt feature
R1	Abstract system model	Abstract RDF system model
R2	System correctness	Atomic execution of safe steps or optimistic scheduling
(R3)	Guarantee of reconfiguration completion	Incremental execution of reconfiguration actions
R4	Interruption to incoming events	Deadline-aware scheduling
(R5)	Meeting of event deadline	Deadline-aware scheduling
R6	No restrictions on number of entities	Deadline-aware scheduling
R7	No constraints on entities	RDF system model
(R8)	No a-priori knowledge of reconfiguration necessary	Deadline-aware scheduling

Table 3.3: Requirements vs. TimeAdapt features

Chapter 4

TimeAdapt Implementation

This chapter describes a Java-based implementation of the TimeAdapt reconfiguration model and its underlying system model. The chapter begins with a detailed overview of the mappings from the abstract system model introduced in the previous chapter to a component-based implementation of reconfigurable actors. The chapter then discusses the main architectural elements of the reconfiguration model, such as the reconfiguration manager and scheduling algorithms. The methods provided by the reconfiguration manager for interacting with a reconfiguration designer as well as the interactions between TimeAdapt itself and the system model elements are then elaborated.

4.1 Architecture Overview

The time-adaptive reconfiguration model described in this thesis was implemented as part of the TimeAct framework. The TimeAct framework is a software framework targeting embedded software and enabling the dynamic reconfiguration of time-dependent software at run-time. It combines an automatic predictive timing model with our time-adaptive reconfiguration model. Figure 4.1 illustrates the overall system architecture of the TimeAct framework. The *Adaptation Manager* is responsible for monitoring the application and triggers adaptations, when specific environmental variables overstep a given threshold. *TimePredict* determines, whether the new configuration conforms to the given timeliness specifications by using measured data and statistical prediction models. This thesis concerns only the TimeAct component model, on which reconfigurations are executed, and the implementation of TimeAdapt. A more detailed description of the TimeAct component model follows in Section 4.2. A detailed description of TimeAdapt itself follows in Section 4.3. For readability, return values and parameters of the used methods are omitted. A detailed summary of the method signatures can be found in Appendices A and B.

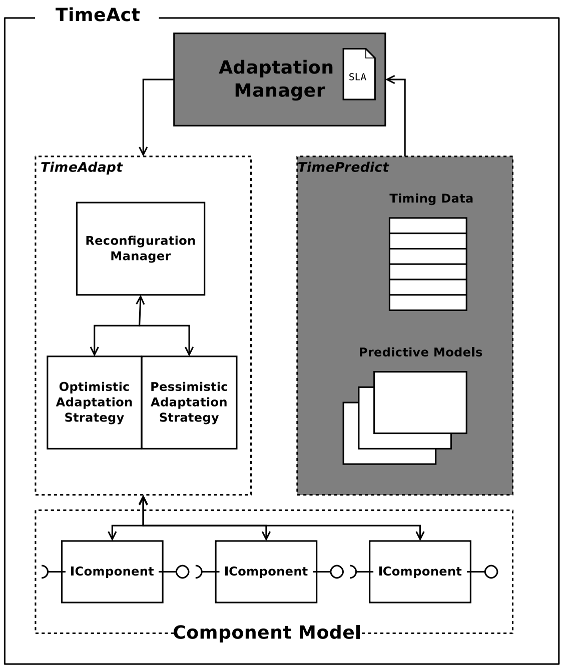


Fig. 4.1: TimeAct System Architecture

4.2 TimeAct Component Model

General-purpose component models for embedded software, such as OpenCom (Coulson et al., 2008), Runes (Costa et al., 2007), OSGI (OSGI Alliance, 2009), or Fractal (David &

Ledoux, 2006a) realise a synchronous computational model, and the LooCI component model (Hughes et al., 2009) realises an event-driven computational model. None of these component models, however, supports embedded software that follows a dataflow-based computational model. Because TimeAdapt is defined for a dataflow-based system model, we implemented a new component model, TimeAct Component Model, on which TimeAdapt was implemented.

The TimeAct component model falls into the category of object-based component models, i.e., components are defined directly by an object-oriented language (Lau & Wang, 2005). The model is implemented in Java, more specifically Java ME (Sun, 2009). The Java implementation enables the applicability of the system model to a wide range of embedded system platforms, from very resource-constrained sensor nodes, to more resourceful embedded PCs. Figure 4.2 illustrates the classes and methods that comprise the TimeAct component model.

The TimeAct Component Model maps actors to implementation units, so-called components. The model supports hierarchical actors, which contain multiple sub-actors. For simplicity, however, we assume for the rest of the discussion that an actor is mapped to exactly a single atomic implementation unit, which does not contain other actors. See Chapter 6.2 for a discussion on the implications of relaxing this constraint on the reconfiguration manager. For the rest of this thesis, the terms actor and component can be used interchangeably.

A component is an active entity that reads data from its input ports, performs a computation based on this data and writes out data on its output ports. Components are connected via channels that connect an input port of one component to the output port of another component. In the current implementation, all components are considered to reside in a single address space and channels connect components with each other.

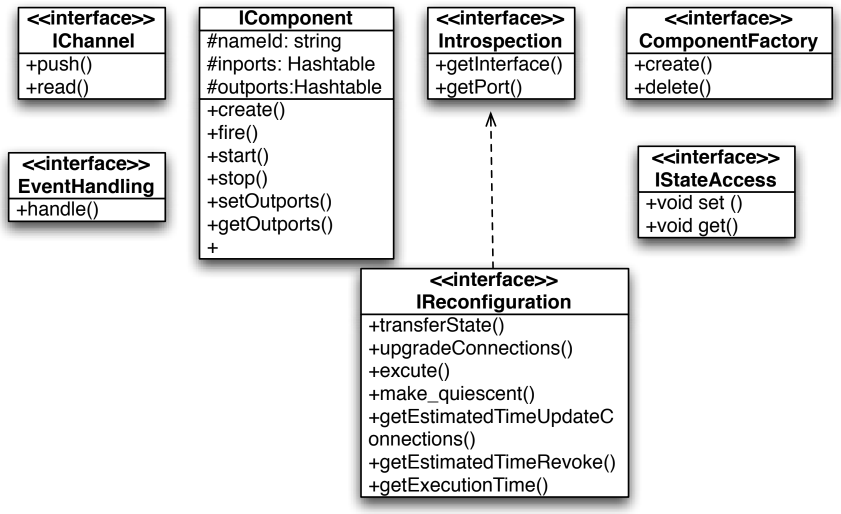


Fig. 4.2: TimeAct Component Model

4.2.1 IComponent

The abstract class `IComponent` is the base class for all component implementations. This base class contains attributes common to all components, such as the set of input and output ports (`Hashtable inports` and `Hashtable outports`) that are realised as Java hashtables. Additionally, this base class contains a unique numeric identifier. The provided abstract operations address the lifecycle of a component. These abstract lifecycle operations include a `fire()` method, as well as a `start()` and a `stop()` method. Implementations of the `fire()` method in a specific component select the specific input ports to read from, compute component-specific values and write these values to selected output ports, all according to a component's functionality. The `start()` method initialises an actor by initialising its respective ports, and the `stop()` method stops an actor by sending remaining calculated data values on the respective output ports.

Additionally, the abstract base class contains accessor methods to set and retrieve connections to other components, via their `setInport()`, `setOutport()`, `getInport()`, `getOutport()` methods.

4.2.1.1 Component Creation and Deletion

Component creation and deletion is part of the normal life cycle of a component and is itself not specific to the TimeAct component model. However, components need to be created and deleted through an external entity that assigns an unique identifier to each component. This external entity needs to implement the `ComponentFactory` interface, which provides methods to create (`create()`) and delete components (`delete()`).

Each concrete component class has a static `create()` method, which is called by the `create()` method of the `ComponentFactory` interface. The `delete()` operation of a component takes as parameter the unique ID of the component and deletes the component from the system configuration.

4.2.1.2 Component Configuration and Runtime

Composition in the TimeAct Component Model takes place during the deployment phase, and is written by the application developer in the form of a `configuration` class. A configuration class is a Java class that creates and connects components to each other and that schedules them for execution. Listing 4.1 illustrates an example configuration with a single component instance and the different lifecycle operations called on this component.

```
public class Configuration{
ComponentFactory cf;
IComponent one;
public Configuration() {
        cf = new FactoryClass();
}
public void run() {
        one = cf.create(ONE.Type);
        cf.register(one);
        one.start();
        while(...) {
```

```
        one.fire();
        }
        one.stop();
        cf.remove(one);
}
public static void main(String[] args) {
        Configuration c = new Configuration();
        c.start();
}
}
```

Listing 4.1: Configuration class

Note that we only consider an implementation running on a single-processor embedded platform. Hence, at each point in time only a single actor thread is active. The order in which multiple actors execute is determined manually by the application developer. At all times it needs to be ensured that no actor blocks due to insufficient data on its input ports.

4.2.1.3 Optional Functionality

Each component must implement the lifecycle operations as they are part of the functional behaviour. Reconfiguration or introspection capabilities are optional and are realised by specific interfaces. The advantage of this approach is that components can be relatively lightweight, implementing only necessary functionality. The following interfaces are supported and discussed in detail in the following subsections.

- Introspection interface, which obtains information about a component's currently implemented interfaces, as well as input and output ports;
- Reconfiguration interface, which allows the execution of behavioural changes on a component;

- State Access interface, which provides methods to obtain and transfer state between stateful components;

- Event Handling interface, which provides methods to react to specific incoming events.

Introspection: The introspection interface provides accessor methods, such as `getInterface()` and `getPort()`. These accessor methods obtain information about a component's currently implemented interfaces, as well as their available input and output ports. The introspection interface is needed to access the reconfiguration interface and its associated methods dynamically at runtime.

Dynamic Reconfiguration: The reconfiguration interface supports the execution of behavioural changes, such as component updates or replacements. Structural changes, such as changes to the component topology, are executed by the reconfiguration manager and are discussed in detail in Section 4.3.1.

Before a reconfiguration starts, all affected components are brought into a reconfiguration-safe state via the method `make_quiescent`. Possible implementations of this method are reader-writer locks or thread-counting (Soules et al., 2003), see Chapter 3, Section 3.2.2.1 for a discussion on reconfiguration-safe states. In this implementation we have chosen the reader-writer lock approach for simplicity. This method ensures that reconfigurations are executed in an isolated way, without interleaving with other actions that are potentially executed on the component.

The method `execute()` realises an actual execution of a reconfiguration action. The implementation of this method depends on the respective reconfiguration action, see Section 4.3.5.2 for a detailed discussion. The execution of a reconfiguration action should be atomic, however, in the current implementation there are no explicit mechanisms provided to guarantee that an `execute()` method either fully completes or is revoked. The methods `transferState()` and `upgradeConnections()` are used by the optimistic scheduling mode. `transferState()` retrieves the values of component variables and transfers these values to a new component for initialisation. `upgradeConnections()` redirects a given input port connection to a new output port, different from the existing one.

The reconfiguration interface provides also the following estimation methods that are needed for the TimeAdapt scheduling algorithms. The `getExecutionTime()` method returns the estimated time of a complete reconfiguration action, used by the pessimistic scheduling algorithm. The `getEstimatedTimeUpdateConnections()` returns an estimate of the connection update time, whereas the `getEstimatedRevokeTime()` returns an estimate of the time it takes to revoke connections. Both methods are used by the optimistic scheduling algorithm.

State Access Interface The state access interface provides accessor methods, such as `set()` and `get()` that access the state of a component. These methods are needed to realise a direct state transfer mechanism between the old and the new component, in which the update mechanism is responsible for extracting and setting the state variables.

Event Handling An event handling interface allows a component to react to incoming events. This is needed by the reconfiguration manager to start the TimeAdapt algorithm. As the type of events is application-dependent, this thesis assumes that the handling of events is implemented by the component designer.

For illustration purposes, Figure 4.3 summarises the use of the different classes and interfaces of the TimeAct component model by means of two user-defined components, `BasicComponent` and `ReconfigurableComponent`. The user-defined components are illustrated in the figure with a diamond. Both components have two port-variables `inport` and `outport`. An example for a connection between these ports is given in the next section.

`BasicComponent` is an example of a component that just provides its own functionality, but no other optional functionality, such as reconfiguration. It extends only from the abstract base class `IComponent` and implements the metods of the `ComponentFactory` interface.

In contrast, ReconfigurableComponent is an example for a component that not only provides its own functionality, but also supports behavioural reconfiguration, i.e., the upgrade or replacement of its implementation. For this, the component implements additionally the methods provided by the Introspection, IReconfiguration, and StateAccess interfaces. Also, this component implements the EventHandler interface, denoting that it can react to specific incoming events.

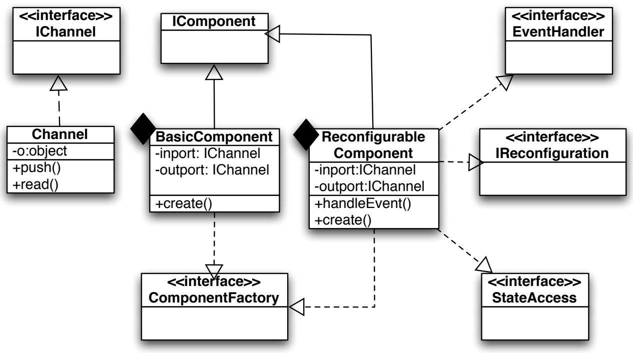


Fig. 4.3: TimeAct Component Model: User-defined classes

4.2.2 IChannel

A component's interface is defined by its set of input and output ports. A port is realised as an entry in a Hashtable that is accessed via its unique name and contains either a Channel object or null if the port is not connected to any other component. A channel represents a FIFO-buffer and provides methods to read and write data of a specific type from or to this channel. Each Channel object needs to implement the IChannel interface and its respective methods, push() and read(). Created channel objects can be set and retrieved from components by using the accessor methods for input and output ports.

Channel objects are created between two components and their respective input and output port. For example, for the two user-defined components in Figure 4.3, a channel between the output port of BasicComponent and input port of ReconfigurableCom-

`ponent` is created via the method call `connect()`. This method call is part of the reconfiguration manager, see Section 4.3.1.

4.3 TimeAdapt Reconfiguration Model

This section describes the implementation of the TimeAdapt reconfiguration model that reconfigures system entities realised with the TimeAct Component Model. Figure 4.4 illustrates a class diagram of the reconfiguration model entities. The core functional system parts of TimeAdapt are the `ReconfigurationManager`, as well as the `ReconfigurationAlgorithm` sub-classes. The `ReconfigurationManager` provides the entry point for reconfigurations and acts as the mediator between the reconfigurable components and the core functional system parts of the reconfiguration model. The `ReconfigurationAlgorithm` classes implement the deadline-aware scheduling algorithms, discussed in Chapter 3, Section 3.2.2.2.

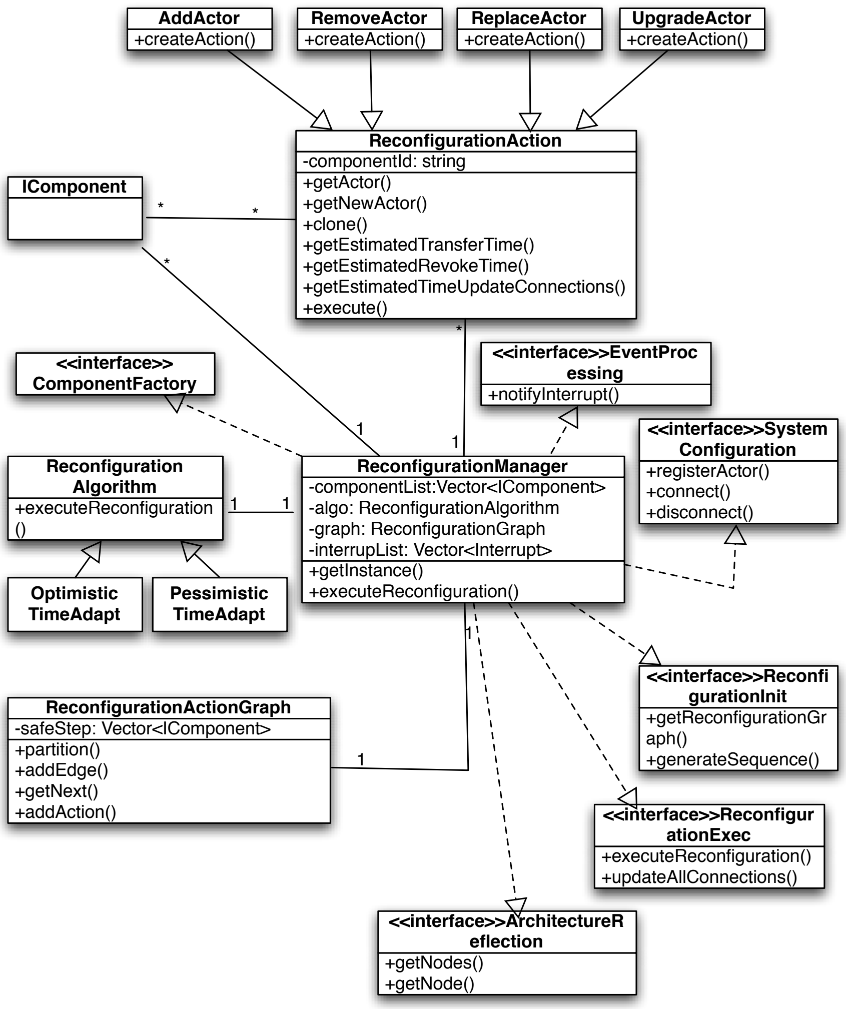

AddActor
+createAction()
RemoveActor
+createAction()
ReplaceActor
+createAction()
UpgradeActor
+createAction()
ReconfigurationAction
-componentId: string
+getActor()
+getNewActor()
+clone()
+getEstimatedTransferTime()
+getEstimatedRevokeTime()
+getEstimatedTimeUpdateConnections()
+execute()
IComponent
*
*
*
*
<<interface>> ComponentFactory
<<interface>>EventProcessing
+notifyInterrupt()
1
1
<<interface>>System Configuration
+registerActor()
+connect()
+disconnect()
Reconfiguration Algorithm
+executeReconfiguration ()
1
1
ReconfigurationManager
-componentList:Vector<IComponent>
-algo: ReconfigurationAlgorithm
-graph: ReconfigurationGraph
-interrupList: Vector<Interrupt>
+getInstance()
+executeReconfiguration()
Optimistic TimeAdapt
Pessimistic TimeAdapt
1
<<interface>>ReconfigurationInit
+getReconfigurationGraph()
+generateSequence()
ReconfigurationActionGraph
-safeStep: Vector<IComponent>
+partition()
+addEdge()
+getNext()
+addAction()
1
<<interface>>ReconfigurationExec
+executeReconfiguration()
+updateAllConnections()
<<interface>>ArchitectureReflection
+getNodes()
+getNode()

Fig. 4.4: TimeAdapt Reconfiguration model implementation

4.3.1 Reconfiguration Manager

A reconfiguration manager is a singleton class that is deployed in every TimeAdapt reconfiguration model instance and is created via a static `getInstance()` operation. This static operation calls the class's constructor of the `ReconfigurationManager` class, in which fields, such as the reconfiguration scheduling algorithm used are instantiated. The current implementation provides an operation that sets the scheduling algorithm used manually, however, it also supports setting it using property files.

The reconfiguration manager maintains a list of all components that are deployed in its address space `Vector<IComponent> componentList`, as well as the currently used scheduling algorithm `ReconfigurationAlgorithm algo`. Additionally, it contains a field denoting the reconfiguration action graph `Graph graph`, as well as a list of all events that have occurred and that still need to be processed `Vector<Interrupt> interruptList`. It also contains a priority queue, in which incoming events are stored in order of their processing deadline `EventQueue queue`.

The reconfiguration manager implements the `ComponentFactory` interface and is responsible for the creation and deletion of components in its address space. Additionally, the reconfiguration manager provides the following interfaces to components and connections:

System Configuration Interface

This interface provides operations to change the current system configuration and to execute structural reconfiguration operations. These include the `registerActor()` method to register a newly created component and methods to connect (`connect()`) and disconnect (`disconnect()`) components to and from the current system configuration.

Architectural Reflective Extension Interface

The architectural reflective extension interface provides operations that return the topology of the current set of components installed in a single address space. This extension provides the `getNodes()` method that retrieves all components that are part of the current system configuration. It also provides the `getNode()` method that retrieves a single component identified by its unique name.

Reconfiguration Initialisation Interface

This interface provides operations that deal with the management of newly created reconfiguration actions into a reconfiguration action graph. The current reconfiguration graph is accessed via the method `getReconfgurationGraph()`. The method `generateSequence()` takes as input a sequence of reconfiguration actions and returns the generated reconfiguration graph. Section 4.3.2 provides a more detailed discussion of the reconfiguration action graph.

Reconfiguration Execution Interface

This interface provides operations that deal with the actual execution of a reconfiguration sequence. The method `executeReconfiguration()` executes all the reconfiguration actions currently contained inside the reconfiguration graph. The method `updateAllConnections()` retrieves all dependent components, i.e., components that are connected to the replaced component via an input port, and updates their input ports accordingly. It is called, for example, when a reconfiguration action needs to be revoked.

Event Processing Interface

This interface provides the `notifyInterrupt()` method that deals with the processing of incoming events. This method is called by any component in the system configuration when it receives an incoming event. This then starts the deadline-aware scheduling algorithm referenced by the field `algo`, which is of type `ReconfigurationAlgorithm`.

4.3.2 Reconfiguration Action Graph

The reconfiguration action graph is an internal data structure and is part of the reconfiguration manager. It contains all reconfiguration actions as a partially ordered set, and is responsible for storing reconfiguration actions in safe steps. Safe steps are realised as container objects that hold a set of reconfiguration actions (`Vector<IComponent>[] safeStepSet`). For this thesis it is assumed that the safe steps are created and initialised by the reconfiguration manager at design time.

The class provides a method to build up temporal dependencies between safe steps (`addEdge()`), as discussed in Section 3.4.2, and to get the next safe step for execution (`getNext()`).

4.3.3 Scheduling Algorithms

The class `ReconfigurationAlgorithm` serves as the abstract base class for the optimistic and pessimistic scheduling algorithms. This class has an abstract `executeReconfiguration()` method, which needs to be overridden by concrete scheduling algorithm implementations.

TimeAdapt contains two algorithm implementations: `OptimisticTimeAdapt` that realises the optimistic scheduling mode and `PessimisticTimeAdapt` that realises the pessimistic scheduling mode. The implementations of both classes follow the algorithms presented in the previous chapter.

4.3.4 Incoming Events

Incoming events are either received by a system model entity or the reconfiguration manager itself by implementing event listener interfaces. The class `Interrupt` realises incoming events with their deadlines and types and is defined as follows, see Listing 4.2:

```
class Interrupt {
private string type;
private double processingDeadline;
public Interrupt(string t, double d) {...}
}
```

Listing 4.2: Interrupt class

4.3.5 Reconfiguration Actions

All reconfiguration actions extend from the abstract base class `ReconfigurationAction`. This class contains the unique identifier of the component on which the reconfiguration action is executed. Additionally, the class contains a `boolean` field that denotes whether the action is an interface-changing or non-interface changing action. This information is needed by the heuristic partitioning algorithm to determine whether the action is partitioned into its own safe step or needs to be partitioned with other actions.

The abstract base class provides methods that deal with the estimation of the overall reconfiguration action, `getEstimatedTime()`, or the estimation of the upgrade-connection phase, `getEstimatedTimeUpdateConnection()`, or revoke time, respectively `getEstimatedRevokeTime()`. In all three cases, the methods forward respective calls to the estimation functions of the component addressed.

Methods that need to be implemented by the concrete sub-classes include accessor methods to retrieve the affected component, such as `getActor()` or methods to retrieve the new component `getNextActor()`. The method `getCurrentPhase()` is used by the optimistic scheduling algorithm to determine the current phase of the reconfiguration action. Also, a concrete sub-class must implement the abstract method `execute()`, which contains the actual implementation of the respective action. A more detailed description of how the `execute()` method is realised for the different actions is presented in subsection 4.3.5.2.

Listing 4.3 summarises the concrete reconfiguration action classes, which represent the available reconfiguration actions identified in Chapter 3, Section 3.2.1. Note that the reconfiguration action classes `Connect` and `Disconnect` are only indirectly realised as actions, since they are part of the reconfiguration manager, or more specifically, part of the reconfiguration manager's system configuration interface.

```
class AddActor()
class RemoveActor()
class ReplaceActor()
class UpgradeActor()
```

Listing 4.3: Implemented reconfiguration action classes

4.3.5.1 Reconfiguration Action Creation

Figure 4.5 illustrates the steps that are executed by a reconfiguration designer at reconfiguration design time, via the `Configuration` class. Note that the following methods are not automated by a tool, but are called by the reconfiguration designer at design time of a reconfiguration.

The reconfiguration designer composes a reconfiguration sequence by firstly creating a set of reconfiguration actions. Each reconfiguration action class implements a static method `createAction()` that returns an instance of the specific reconfiguration action type. This method requires different parameters, depending on the type of the reconfiguration action. A `RemoveActor` action is created by calling the static `createAction(String nameId)` method, which accepts the unique identifier `nameId` of the component to be removed as parameter. This method assumes that the identified component is part of the current system configuration, otherwise an exception is thrown. An `AddActor` action is created by calling the static `createAction(TypeId typeId)` method, which accepts the type `typeId` of the component to be created as parameter. This then calls the respective `create()` method of the component it-

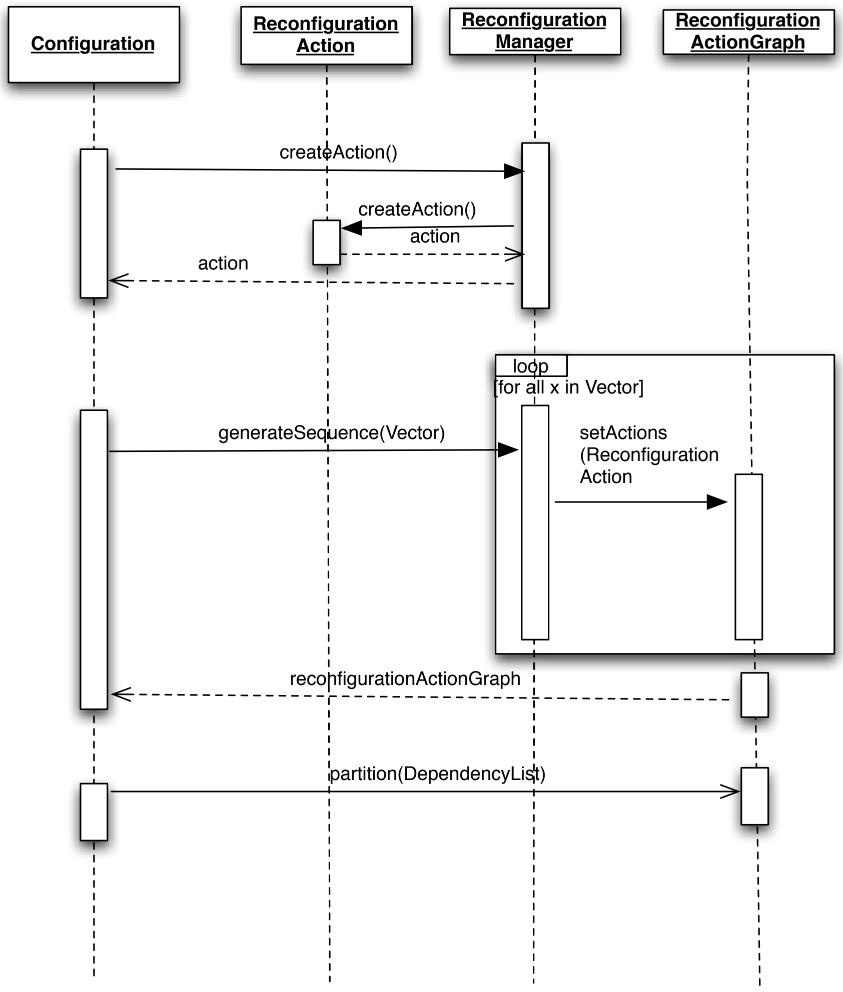


Fig. 4.5: Reconfiguration sequence generation

self. `ReplaceActor` and `UpgradeActor` actions are created by calling the static method `createAction(String nameId, TypeId newActorId)`, with the unique identifier `nameId` of the component to be replaced and the `typeId` of the newly replaced component as parameters.

After all reconfiguration actions are created, the reconfiguration graph structure is initialised with these actions by calling the `generateSequence()` method. The subsequent reconfiguration action partitioning is realised by calling the `partition()` method, provided by the `ReconfigurationActionGraph` object. This method implements the reconfiguration partitioning heuristic, explained in Chapter 3, Section 3.4.1. As a parameter, the method requires the dependency description of the actors that are part of the system configuration. `DependencyList` contains, for each actor, an instance of the `Dependency` class, where class `Dependency` is defined as:

```
class Dependency {
        IComponent actor;
        Hashtable<IComponent> dependentActors;
}
```

Listing 4.4: Dependency class

The result of the partitioning is a total ordered set of the involved reconfiguration actions in safe steps.

4.3.5.2 Reconfiguration Action Runtime

The actual execution of reconfigurations is realised by the reconfiguration execution interface of the reconfiguration manager and the `IReconfiguration` interface provided by reconfigurable components. Figure 4.6 illustrates the interactions between the reconfiguration manager and other entities when no incoming event occurs during reconfiguration.

A reconfiguration is triggered by calling the `executeReconfiguration()` method of the reconfiguration manager. Reconfigurations can be either triggered explicitly by a reconfiguration designer or can be triggered by changes in the operating conditions of the system. However, the reconfiguration manager is oblivious to the cause of the trigger. The `executeReconfiguration()` method gets as parameter the list of reconfiguration actions to execute. The reconfiguration manager then retrieves the next safe step to execute by calling the `getNext()` method of the `ReconfigurationActionGraph` class.

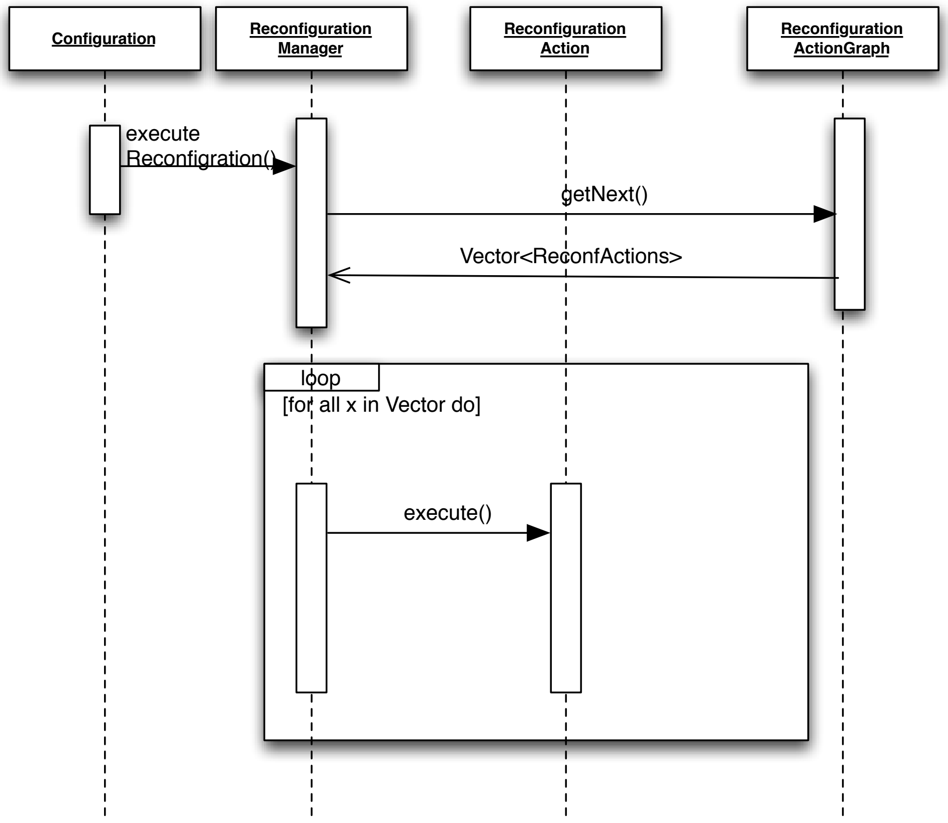


Fig. 4.6: Non-interrupted reconfiguration execution

This method returns all the actions that need to be executed atomically to reach a safe system configuration. The reconfiguration manager then loops sequentially over the list of actions in the safe step and calls their respective `execute()` method. This method is realised differently in the respective reconfiguration action sub-classes.

The `AddActor` reconfiguration action class realises the `execute()` method by calling the static `createActor` method of the reconfiguration manager, with the type identifier of the component to be added as parameter. The `RemoveActor` action class contains the unique identifier of the actor to be removed. The implementation of `execute()` results in a call to the `removeActor()` method of the reconfiguration manager, that then executes the actual object removal. `Replace` and `Upgrade` reconfiguration actions contain the identifier of the actor to be replaced and the type identifier of the replacement actor. In both cases, the `execute()` method first creates the component, which replaces the old component, by calling the static `createActor()` method of the reconfiguration manager. If the component to be replaced was stateful, a state transfer is executed by calling the `transferState()` method on the component to be replaced. Afterwards, the method `upgradeConnections` from the reconfiguration manager is called, which upgrades all components that have connected input ports to point to the newly replaced component. The reconfiguration actions `Connect` and `Disconnect` are realised by the reconfiguration manager. `Connect` is realised by calling the `connect()` method of the manager, which takes as parameters the two components to be connected and their input and output ports. Likewise, the `disconnect()` method of the manager takes as parameters the components to be disconnected and their ports.

Figure 4.7 illustrates the interactions between the reconfiguration manager and the entities involved when an incoming event is received, for example by the `EventHandler` interface of a component, and there is no other event currently processed by TimeAdapt. In this case, the operation `notifyInterrupt()` of the reconfiguration manager is called, with the event processing deadline as parameter. The reconfiguration manager then calls the `executeReconfiguration()` method of the chosen scheduling algorithm, with the event processing deadline and the remaining reconfiguration action graph as parameters. Depending on the scheduling algorithm, the current safe step is retrieved (pessimistic mode), or the phase of the currently executing reconfiguration action is retrieved (optimistic mode). Based on the given deadline, the algorithm then decides how to proceed. If an incoming event is received, and TimeAdapt is already processing another event, the event is queued in the event queue, according to its associated processing deadline.

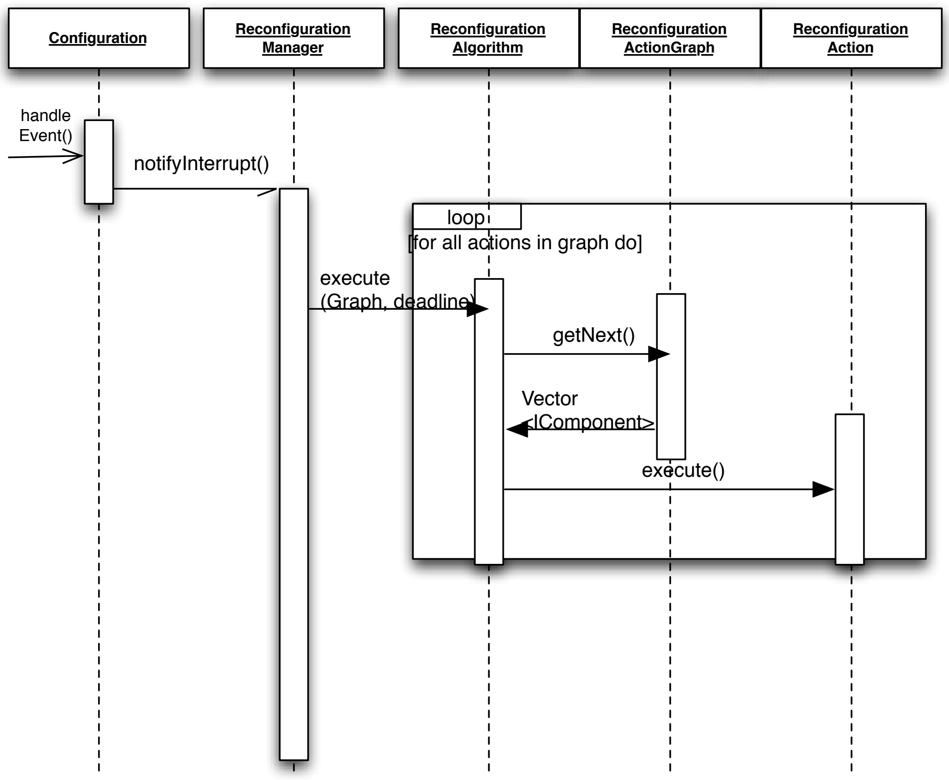


Fig. 4.7: Interrupted reconfiguration execution

4.4 Summary

This chapter described the Java-based implementation of the system model and the reconfiguration model as defined by their design presented in Chapter 3. First the TimeAct Component Model was presented, which is a direct mapping of the abstract dataflow based system model to a Java-based implementation of reconfigurable actors. The component model was implemented for this work because there are no component models that target the dataflow based computational model. The TimeAct component model realises a layered approach, in which interfaces realise different functional and non-functional concerns of a component, such as reconfiguration. The system elements of the TimeAdapt reconfiguration model implementation were then presented. The central part of this implementation is the reconfiguration manager, which has access to all components and executes reconfigurations on these components. The manager implements a series of interfaces that deal with the generation of reconfiguration actions as well as the actual execution of structural reconfigurations on the current system configuration. Finally, the usage of the reconfiguration model by a reconfiguration designer, and the interaction between TimeAdapt itself and its involved entities during reconfiguration execution were outlined.

Chapter 5

Evaluation

This chapter presents an evaluation of TimeAdapt as a realisation of a time-adaptive reconfiguration model. First it presents the evaluation objectives, and the metrics used to measure the performance of TimeAdapt. The main focus of this chapter is on the experiments used for the evaluation, as well as their analysis and outcomes.

5.1 Objectives

Chapter 2 outlined the issue with existing reconfiguration models for embedded software. A transactional reconfiguration model only processes incoming events when a reconfiguration is completed, whereas a pre-emptive reconfiguration model cannot guarantee the progress towards reconfiguration completion, especially in the case of a high incoming event rate. The previous chapters 3 and 4 described the design and implementation of TimeAdapt, a reconfiguration model designed to react to incoming events in a timely fashion, while at the same time, making progress towards a reconfiguration completion. TimeAdapt's evaluation assesses to which degree requirements are met when the model is applied on a realistic embedded system platform.

The following performance objectives are used:

O1) TimeAdapt meets more event deadlines than a transactional reconfiguration model under varying environmental conditions, such as event arrival rate, event processing deadlines, action execution durations, and number of event sources.

O2) TimeAdapt has at most the same number of remaining reconfiguration actions as a preemptive reconfiguration model when an event is processed. This holds for varying event arrival rates, event processing deadlines, action execution durations, and number of event sources.

O3) Due to its more complex scheduling mechanism, the time-adaptive reconfiguration model has an inherent execution time overhead.

The next section presents the metrics used to assess the performance of TimeAdapt against these objectives.

5.2 Metrics

The rationale for the metrics used comes from two domains, namely component-based software reconfiguration approaches and scheduling algorithms. As performance goal O1 is to measure timeliness of the reconfiguration model, we measure the percentage of event processing deadlines that are met, similarly to the domain of scheduling algorithms (Liu, 2000). The percentage of remaining reconfiguration actions is a metric that indicates the progress a reconfiguration model makes towards reconfiguration completion, i.e., how much of a given reconfiguration sequence has been executed. In addition, the total reconfiguration time is a widely used metric when measuring the performance overhead of reconfiguration (Rasche & Polze, 2003), (Dowling, 2004).

- Percentage of event processing deadlines met. This metric is calculated by measuring the time elapsed between the occurrence of an interrupting event and the time at which processing of this interrupted event is started and comparing this time against the event's processing deadline. The percentage of event processing deadlines met is an indicator of how timely a model reacts to incoming events.

A lower percentage rate of processing deadlines met means a poorer timeliness performance with regards to incoming events.

- Percentage of remaining reconfiguration actions. The percentage of remaining reconfiguration actions is calculated from the average number of remaining reconfiguration actions in all experiment runs and the reconfiguration sequence length. A lower percentage of remaining reconfiguration actions indicates a faster reconfiguration completion.
- Overall reconfiguration duration. This metric measures the time elapsed between reconfiguration start and reconfiguration completion.

5.3 Experiments

A set of actors has been implemented with the TimeAct component model. This set of actors realises an embedded sensing scenario, as described in Figure 5.1. The scenario was chosen as it represents the type of applications that would benefit from a time-adaptive reconfiguration model, see Chapter 3.2.1. In this scenario, a temperature sensor actor calculates the current temperature based on the sensor data it receives from its temperature sensor. A control actor activates an alert actor if the temperature is over a specified threshold. The alert actor then sends the respective temperature data to an output actor that then either forwards the temperature data to an external entity or outputs it on a central monitor. Events in this application are represented by interrupts from the underlying hardware.

5.3.1 Hardware and Software Configuration

The following configuration of hardware and software was used: The experiments were performed on an embedded device platform, Java SunSpots, which are small, wireless sensing devices (Sun, 2006). Even though Java SunSpots are more powerful in terms of

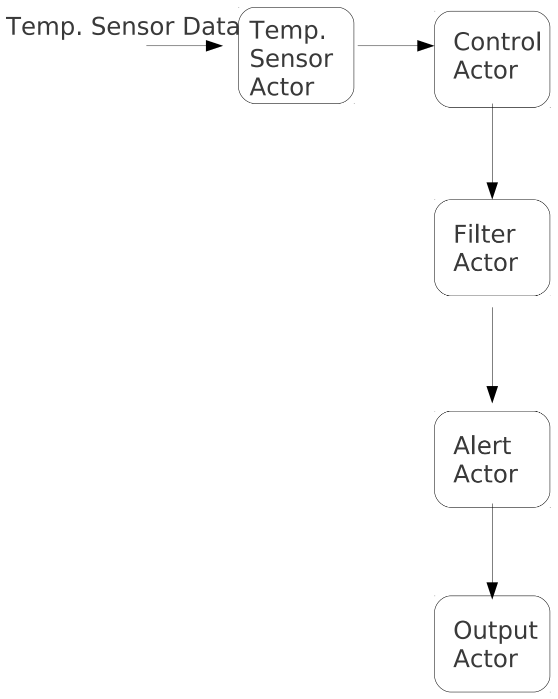


Fig. 5.1: Temperature sensor scenario realised on embedded platform

resources than many other embedded platforms, such as Motes (Crossbow Technology Inc., 2004), the rationale for using them is twofold: Firstly, there are many examples of software that would benefit from a time-adaptive reconfiguration model running on this platform. Examples include sensing and monitoring applications (Hughes et al., 2010). Secondly, in contrast to other platforms, existing solutions for this platform provide comparative values for typical reconfiguration action times. We base the execution durations for reconfiguration action types on values obtained by the LooCi component model that was evaluated on the same platform (Hughes et al., 2009).

All time measurements were taken using the `AT91 Timer Counter` integrated time capture functions, which are part of the embedded processor board (Goldman, 2009). The time capture function supports the execution of periodic tasks, such as the raising of interrupts.

The software is realised with the TimeAct component model, our implementation of a dataflow-driven system model. The implementation conforms to the Java ME Connected Limited Device Configuration (CLDC) 1.1 standard (Sun, 2009). TimeAdapt itself implements the pessimistic and optimistic scheduling algorithms as described in Section 3.4.2. For comparison purposes, we implemented the experiments using a transactional reconfiguration model, realised for the dataflow-based system model. This is a modified version of the abstract Kramer and Magee reconfiguration model (Kramer & Magee, 1985). The original abstract model was previously realised for processes in distributed systems (Kramer & Magee, 1990), a realisation we adapted to handle more fine-grained software entities based on the dataflow system model. The implementation processes incoming events only after all reconfiguration actions are completed. Again for comparison purposes, we implemented a modified version of DynaQoS-RDF's preemptive reconfiguration model, defined on TimeAct software components (Zhao & Li, 2007b). In contrast to DynaQoS-RDF's reconfiguration model, our implementation supports stateful actors. The implementation processes incoming events directly.

5.3.2 Parameters

A number of parameters affect the performance of the individual experiments. The parameters' value range used resembles real embedded software, executed on the specific embedded platform. Values from existing work that evaluates software on the SunSpot platform were used, as well as values taken from the platform itself. The parameters are:

- Reconfiguration sequence length l. This parameter determines the number of reconfiguration actions in a reconfiguration sequence. Values for reconfiguration sequence length are specific to the experiment used and are discussed for each experiment individually.
- Reconfiguration action type a. This parameter determines the type of actions that are part of a reconfiguration sequence. Actions vary in their overall execution

duration, and are taken from the LooCI component model (Hughes et al., 2009). The execution durations of specific actions are discussed in more detail in each experiment's section.

- Event arrival rate λ [ms]. This parameter denotes the time rate at which events are generated. We decided to use hardware interrupts, generated from the SunSpot platform, as a source of incoming events for the application. The SunSpot platform provides the `AT91 Timer Counter` class, which allows the periodic triggering of clock interrupts according to different rates (Goldman, 2009). The rate at which the timer counts is determined by the type of clock used. The available clock speeds are:
  - Low clock speed, i.e., duration of 2000 ms until interrupt event is raised.
  - Medium clock speed, i.e., duration of 35 ms until interrupt event is raised.
  - High clock speed, i.e., duration of 8 ms until interrupt event is raised.
  - Very high clock speed, i.e., 2 ms until interrupt event is raised.
- Event processing deadline t_d [ms]. This parameter denotes the deadline associated with events, and indicates the duration within which the event should be processed. Similar to the reconfiguration length parameter, values for event processing deadlines are specific to the experiment used and are discussed for each experiment individually. In general, we abbreviate event processing deadlines as deadlines.

Before each experiment run, events are uniformly generated over a given time interval. The time interval is chosen to be at least the same length as a reconfiguration sequence to ensure that events coincide with reconfiguration actions. The pre-creation of events has the advantage that different reconfiguration models can be executed on the same sequence, enabling easier comparison. A single experiment run returns the execution duration of the applied reconfiguration model, averaged over all occurring events, and the percentage of remaining reconfiguration actions. Each experiment run is repeated

100 times. The timeliness of an experiment is calculated using the average execution duration of ten experiment runs as a resulting execution duration value. This ensures that the results are not influenced by extreme operational conditions.

Table 5.1 summarises the five experiments conducted, including a short description of each experiment. The performance objectives addressed are given in parentheses. In the next sections, the results of these experiments are analysed to assess the extent to which the performance objectives, identified in Section 5.1, are addressed.

No.	Name	Description
1	Uniform Reconfiguration	A single reconfiguration action type with one event source. Four event arrival rates: Low, medium, high, very high, each with varying deadlines (O1, O2).
2	Heterogeneous Reconfiguration	Different types of reconfiguration actions with varying execution durations. Single event source, nne low event arrival rate, with varying deadlines (O1, O2).
3	Varying Safe Step Size	Different types of reconfiguration actions with varying number of actions in a safe step. Single event source, one low event arrival rate, with varying deadlines (O1, O2).
4	Multiple Event Sources	A single reconfiguration action type with two event sources. Medium event arrival rate, homogeneous and heterogeneous deadlines on the events from both sources (O1, O2).
5	Overhead	Execution duration comparison of reconfiguration sequence comprised of single reconfiguration action type, but different event arrival rates. Three event arrival rates: Low, Medium, High, each with the same deadline (O3).

Table 5.1: Summary of TimeAdapt evaluation experiments

5.3.3 Experiment 1: Uniform Reconfiguration

This experiment evaluates TimeAdapt's performance in comparison to a transactional and a preemptive reconfiguration model (RM) under varying event arrival rates and event processing deadlines. Table 5.2 lists the parameter settings used. The reconfiguration sequence used is comprised of homogeneous *upgradeActor* reconfiguration actions that have all approximately the same execution duration, and which upgrade all the components in the scenario. In detail, the reconfiguration actions are:

Parameter	Name	Value
Reconfiguration sequence length	l	5
Reconfiguration action type	a	upgradeActor
Event arrival rate [ms]	λ	2, 8, 35, 2000
Event processing deadline [ms]	t_d	0.15...100

Table 5.2: Experiment 1: Parameter Setting

- *upgradeActor(tempSensor)*
- *upgradeActor(Control)*
- *upgradeActor(Filter)*
- *upgradeActor(Alert)*
- *upgradeActor(Output)*

The execution duration of a single *upgradeActor* reconfiguration action is randomly chosen between 15 ms and 25 ms [1], as used in the LooCi component model evaluation (Hughes et al., 2009). As upgrade actions are non-interface changing, each action denotes its own safe step.

[1]using the Java Random API

The experiment uses a single event source and event arrival rates are generated by the `AT91 Timer Counter`. For this experiment, we use all four event arrival rates: low (2000 ms), medium (35 ms), high (8 ms), very high (2 ms). The actual duration for event processing is ignored, i.e., it is assumed that events can be processed in a negligible execution duration. The experiment is executed for an increasing event deadline from 0.15 ms to 100 ms in steps of 10 ms. The deadline boundaries were chosen as they represent extreme operational settings: 0.15 ms is the minimum time span needed to react to an internal event (Simon et al., 2006), whereas a deadline of 100 ms fits almost the entire reconfiguration sequence.

5.3.3.1 Reconfiguration Timeliness

In this part of the experiment, the percentage of deadlines met for both optimistic and pessimistic modes of TimeAdapt are compared with the implementation of a transactional reconfiguration model. We do not consider the percentage of deadlines met for the preemptive reconfiguration model, as it directly processes incoming events and will always meet an event's deadline. It is expected that TimeAdapt will meet a higher percentage of deadlines compared to the transactional reconfiguration model, as it reacts faster to incoming events. In TimeAdapt, an event's processing deadline is missed when an event is raised and the execution duration of the currently executing safe step or reconfiguration action exceeds the deadline. In all other cases, TimeAdapt should meet the deadline.

Table 5.3 summarises the percentage of deadlines met for three event arrival rates (low, medium, and high), a deadline of 20 ms (low), 50 ms (medium), and 100 ms (high), for both pessimistic and optimistic modes, and the transactional reconfiguration model. Table 5.4 lists the figures that provide more detail of these numbers. The detailed results can be found in Appendix C. Figures 5.2, 5.3, and 5.4 illustrate the results for the three event arrival rates, respectively, with deadlines in the range from 0.15 ms to 100 ms. Figure 5.5 illustrates a zoomed-in view on the results for deadlines ranging between 0.15

ms and 20 ms and a medium event arrival rate. Figure 5.6 illustrates a zoomed-in view on the results for deadlines between 0.15 ms and 20 ms, and a very high event arrival rate. The zoomed-in views are included to illustrate extreme parameter settings (i.e., very small event processing deadlines, and very high event arrival rates) that give an indication of the boundaries of TimeAdapt's benefits.

	Low Arrival Rate			Medium Arrival Rate			High Arrival Rate		
Deadline [ms]	20	50	100	20	50	100	20	50	100
Pessimistic TA	81%	99%	99%	81%	99%	99%	80%	99%	99%
Optimistic TA	99%	99%	99%	99%	93%	82%	67%	71%	48%
Transactional	8%	29%	67%	1%	11%	50%	0%	0%	9%

Table 5.3: Percentage of deadlines met

Scenario	Figure
Low arrival rate; t_d between 0.15 and 100 ms	Fig. 5.2
Medium arrival rate; t_d between 0.15 and 100 ms	Fig. 5.3
High arrival rare; t_d between 0.15 and 100 ms	Fig. 5.4
Zoomed-in medium arrival rate; t_d between 0.15 and 20 ms	Fig. 5.5
Zoomed-in very high arrival rate; t_d between 0.15 and 20 ms	Fig. 5.6

Table 5.4: Mapping between figures and experiment settings

In general, both of TimeAdapt's modes meet a higher percentage of deadlines than the transactional reconfiguration model. When the event arrival rate is low, the transactional model meets an increasing percentage of deadlines when the deadline itself increases. The outliers at 30 ms and 90 ms can be explained by variations in the experiment runs, especially the point in time when an event occurs. An incoming event's deadline is only met if the event occurs towards the completion of the reconfiguration sequence, and the execution duration of the remaining reconfiguration sequence falls within this deadline (see Figure 5.2). With an increasing event arrival rate, more events potentially occur at earlier stages of the reconfiguration sequence and their processing

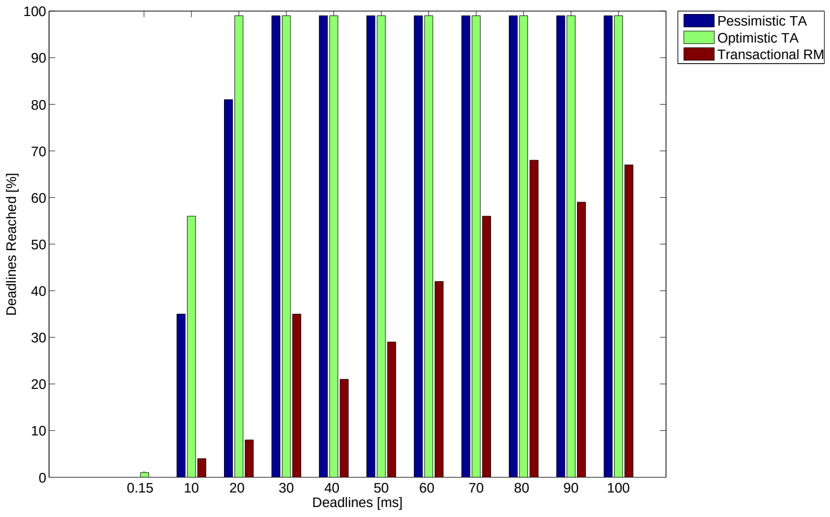


Fig. 5.2: Percentage of deadlines met for low event arrival rate and deadlines range from 0.15 ms to 100 ms

needs to wait for previous events to be completed. In this case, the transactional model meets a lower percentage of deadlines than the two modes of TimeAdapt (see Figures 5.3 and 5.4).

Similarly, in the pessimistic mode, an event's deadline is met if the execution duration of the currently executing safe step falls within the given deadline. Higher deadlines increase the likelihood that an event's deadline is within the execution duration of a safe step. Therefore, the percentage of deadlines met increases with an increasing deadline size for all three event arrival rates.

An increasing event arrival rate leads to a higher likelihood that events occur at the same time. As only one event can be processed by TimeAdapt at a given time, a higher number of events are queued. A queued event meets its deadline if its waiting time, i.e., the time period between its occurrence and its actual processing time, is smaller than the deadline. A smaller event arrival rate implies a smaller number of queued events and a smaller waiting time for each queued event. Therefore, for a given deadline, the

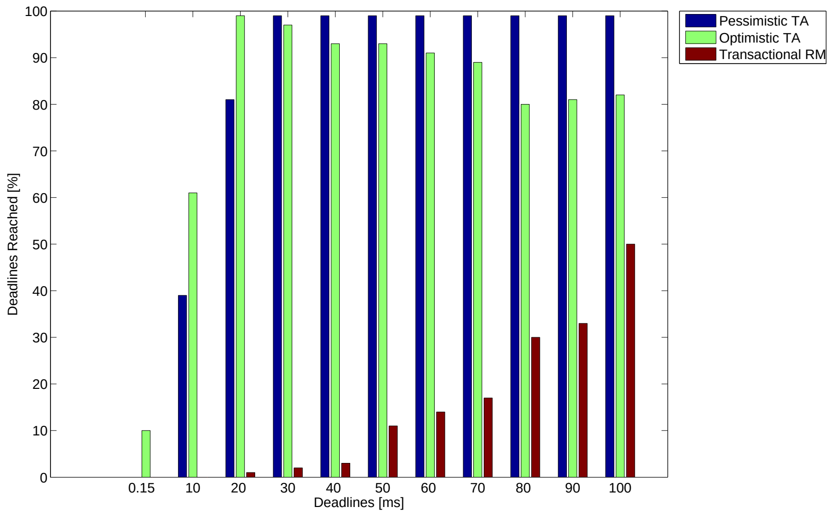


Fig. 5.3: Percentage of deadlines met for medium event arrival rate and deadlines range from 0.15 ms to 100 ms

percentage of deadlines met is higher for lower event arrival rates (see Figure 5.2 vs. Figure 5.3, and Figure 5.3 vs. Figure 5.4).

In TimeAdapt's optimistic mode, a deadline is reached if the completion or undo time of the currently executing reconfiguration action, i.e., either its state-transfer or update-connection phase, is completed within the given deadline. The optimistic mode outperforms the pessimistic mode if the completion or undo time of the currently executing phase is short and falls within the given deadline, even though the completion of the overall reconfiguration action exceeds the deadline.

For the given action execution duration, this is the case if deadlines are smaller or equal to 20 ms (see Figures 5.2, 5.3, and 5.4). With an increasing deadline and a higher event arrival rate, the optimistic mode meets a smaller percentage of deadlines. This is a result of its more complex scheduling. The fine-granular scheduling of the optimistic mode based on reconfiguration action phases increases also the waiting time of queued events. This waiting time increases further with a higher event arrival rate. Therefore,

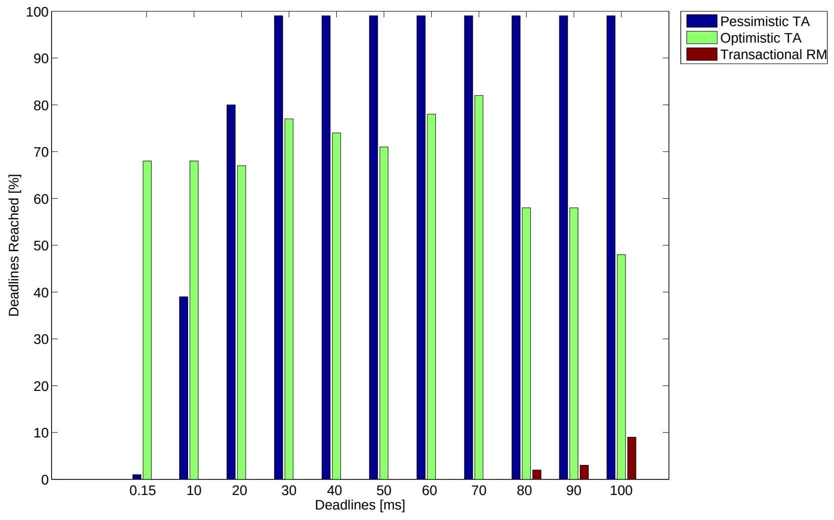


Fig. 5.4: Percentage of deadlines met for high event arrival rate and deadlines range from 0.15 ms to 100 ms

the percentage of deadlines met when using the optimistic mode is smaller for higher event arrival rates (see Figures 5.2 vs 5.3 and 5.3 vs. 5.4).

Figures 5.5 and 5.6 illustrate, in more detail, the difference between the pessimistic and optimistic modes and the transactional reconfiguration model, for a deadline range from 0.15 ms to 20 ms. In this range, the processing deadline is smaller than the execution duration of a single reconfiguration action. Figure 5.5 shows the results for the medium event arrival rate, and Figure 5.6 shows the results for a very high event arrival rate.

For both event arrival rates, the transactional reconfiguration model does not meet any deadline. This is because deadlines are smaller than the execution duration of a single reconfiguration action.

The pessimistic mode always has to complete at least the currently executing safe step, which in this experiment takes the execution duration of a single reconfiguration action. As a result, this mode is more likely to act like a transactional reconfiguration

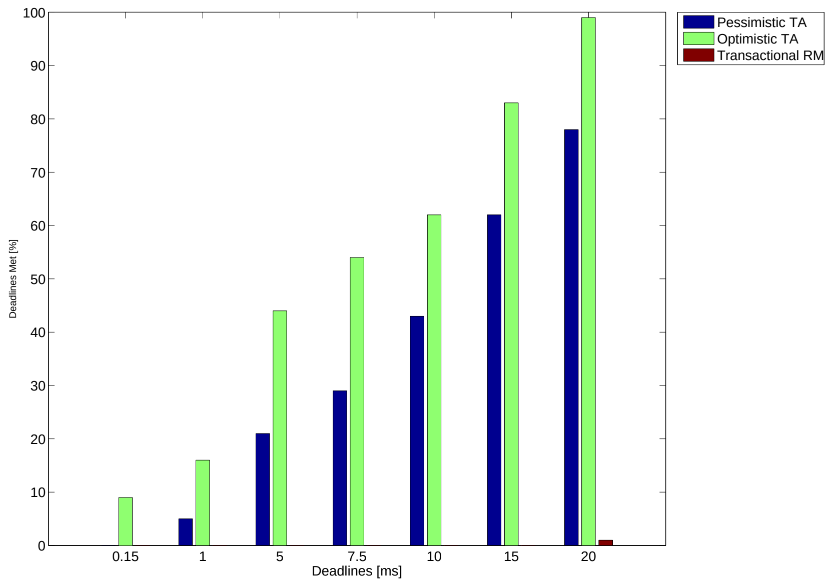


Fig. 5.5: Percentage of deadlines met for medium event arrival rate and deadlines range from 0.15 ms to 20 ms

model, with respect to the percentage of deadlines met, when the deadlines are small.

For both event arrival rates, the optimistic mode meets a higher percentage of deadlines than the pessimistic mode. As already pointed out, this is because reconfiguration action scheduling is more fine-grained than in the pessimistic mode. In the case of a high event arrival rate and a small deadline, the optimistic mode meets almost the same percentage of deadlines than a preemptive reconfiguration model, i.e., all events meet their deadlines (see Figure 5.6). The preemptive behaviour of the optimistic mode is caused by small deadlines, which do not allow completion of any phase. The optimistic mode directly aborts the currently executing phase, and processes queued events more quickly. Note that this assumes a negligible processing execution duration of the event itself.

In summary, the performance of TimeAdapt's scheduling modes with respect to

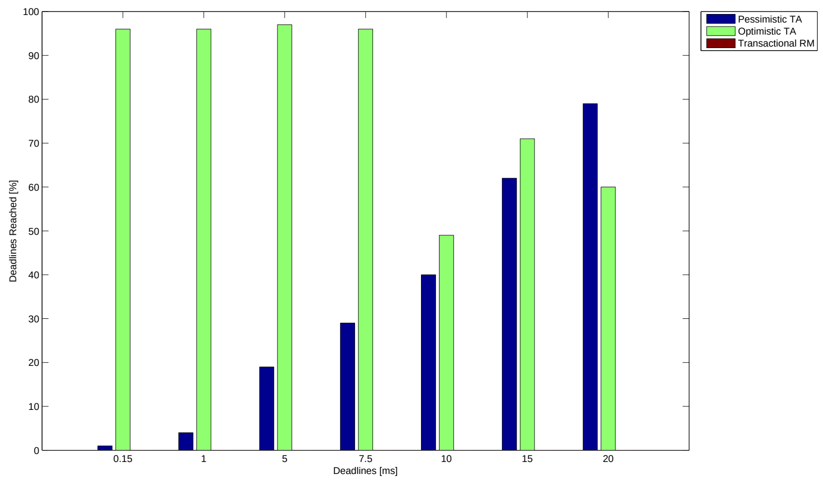


Fig. 5.6: Percentage of deadlines met for a very high event arrival rate and deadlines range from 0.15 ms to 20 ms

meeting deadlines under varying event arrival rates depends on the deadlines allowed by a system. When there are high event arrival rates and small deadlines, i.e., smaller than the actual reconfiguration action execution duration, the optimistic mode shows better performance than the pessimistic mode. In this case, the pessimistic mode always oversteps the deadline, as it needs to complete the currently executing reconfiguration action. When there are medium and low event arrival rates and high deadlines, the pessimistic mode outperforms the optimistic mode. In this case, the pessimistic mode has a smaller scheduling time and as a result a smaller waiting time for queued events. See also Chapter 3.4.2.4 for a detailed discussion of the timeliness behaviour of both modes. In overall, TimeAdapt shows a better overall performance with respect to meeting deadlines than the transactional model, fulfilling objective O1. However, TimeAdapt cannot meet all incoming event's deadlines because an ongoing reconfiguration is prioritised over an incoming event. As a result, embedded software that needs strong guarantees on events, such as hard-real time software, should apply a preemptive reconfiguration model.

5.3.3.2 Reconfiguration Progress

This part of the experiment compares the percentage of reconfiguration actions remaining after an event is processed, for both modes of TimeAdapt and the preemptive reconfiguration model, described in Section 5.3.1. As the transactional reconfiguration model always completes an ongoing reconfiguration before processing any event, we do not consider this model in this section. It is expected that TimeAdapt will have a lower percentage of reconfiguration actions remaining than the preemptive reconfiguration model, indicating an overall faster progress towards reconfiguration completion.

Table 5.5 summarises the percentage of reconfiguration actions remaining for three event arrival rates (low, medium, and high), for a given deadline of 20 ms (low), 50 ms (medium), and 100 ms (high), when using TimeAdapt's pessimistic and optimistic modes, as well as the preemptive reconfiguration model. Table 5.6 lists the figures that show a graphical representation of the results for various experiment parameters. Figures 5.7, 5.8, and 5.9 illustrate the results for low, medium, and high event arrival rates and deadlines in the range from 0.15 ms to 100 ms. Figures 5.10 and 5.11 illustrate more detailed results on the number of reconfiguration actions remaining in the case of two consecutive events, for both modes of TimeAdapt and a medium event arrival rate. Figures 5.12 and 5.13 illustrate the number of reconfiguration actions remaining, when there are two consecutive events, for both modes of TimeAdapt and a very high event arrival rate.

	Low Arrival Rate			Medium Arrival Rate			High Arrival Rate		
Deadline [ms]	20	50	100	20	50	100	20	50	100
Pessimistic TA	42.38%	28.22%	5.06%	43%	25.1%	6.06%	40.70%	26.49%	3.09%
Optimistic TA	52.10%	37.96%	4.62%	49%	34.20%	6.2%	50.71%	39%	14.42%
Preemptive RM	65.36%	60%	60.66%	63%	59%	62%	60%	60%	59%

Table 5.5: Percentage of reconfiguration actions remaining

The preemptive reconfiguration model shows a varying percentage of remaining re-

Scenario	Figure
Low arrival rate; t_d between 0.15 and 100 ms	Fig. 5.7
Medium arrival rate; t_d between 0.15 and 100 ms	Fig. 5.8
High arrival rate; t_d between 0.15 and 100 ms	Fig. 5.9
More detailed view on medium arrival rate; t_d between 0.15 and 100 ms	Fig. 5.10, Fig. 5.11
More detailed view on very high arrival rate; t_d between 0.15 and 100 ms	Fig. 5.12, Fig. 5.13

Table 5.6: Mapping between figures and experiment settings

configuration actions for all three event arrival rates. However, for all three event arrival times, the percentage of reconfiguration actions remaining remains roughly constant throughout all deadlines (see Figure 5.7, 5.8, and 5.9). This is because in a preemptive reconfiguration model, an ongoing reconfiguration is interrupted to process an incoming event. In the experiment, events are uniformly distributed over the reconfiguration sequence, with half of the incoming events occurring at the beginning of a reconfiguration sequence, i.e., either in the first or second reconfiguration action. When an event is processed, approximately 60% of the reconfiguration actions are remaining.

TimeAdapt's pessimistic mode has an increasing percentage of reconfiguration actions remaining for deadlines smaller than 20 ms, which decreases for deadlines equal to or larger than 20 ms when the event arrival rate is low (see Figure 5.7). The increase is because for a very small deadline, the pessimistic mode has to at least complete the currently executing action. However, for higher deadlines, depending on when an incoming event occurs, the mode can abort a further scheduling of actions. For medium and high event arrival rates, TimeAdapt's pessimistic mode has a constant percentage of reconfiguration actions remaining for deadlines smaller than 20 ms and a decreasing percentage for deadlines equal to or larger than 20 ms (see Figures 5.8 and 5.9). The increase or maintenance of the percentage of remaining actions, for deadlines smaller than 20 ms, is because the scheduler does not schedule any further reconfiguration actions, as the deadline is smaller than the execution duration of any safe step. When the

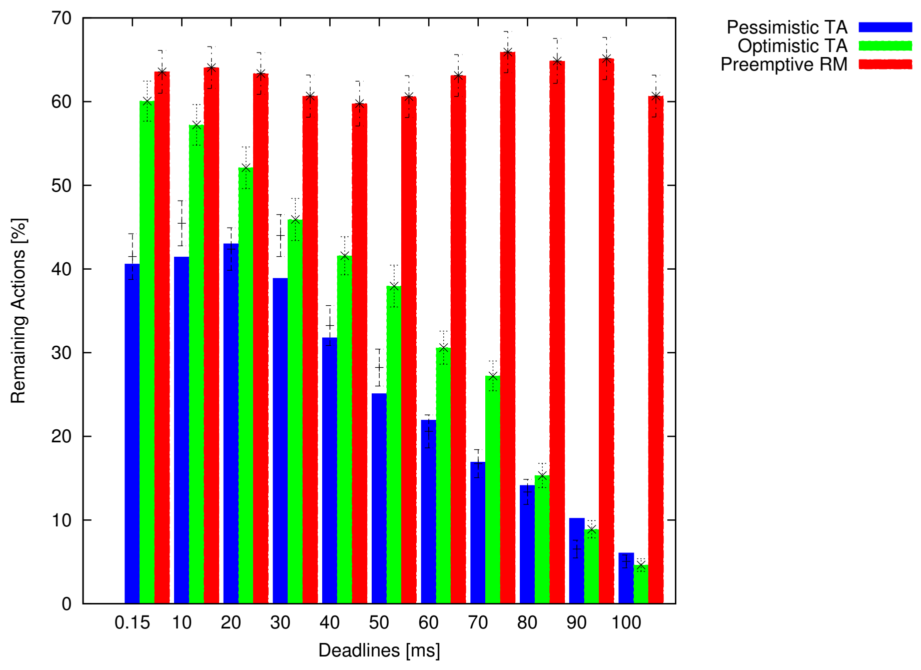


Fig. 5.7: Percentage of remaining actions for low event arrival rate

deadline increases, additional reconfiguration actions can be scheduled, explaining the higher throughput of reconfiguration actions.

TimeAdapt's optimistic mode shows similar behaviour to the pessimistic mode. For all three event arrival rates, TimeAdapt's optimistic mode has a decreasing percentage of reconfiguration actions remaining when the deadline increases (see Figure 5.7, 5.8, and 5.9) Like in the pessimistic mode, the higher percentage of reconfiguration actions remaining for lower deadlines, such as 0.15 or 10 ms, is caused by fact that the mode does not allow the scheduling of additional reconfiguration actions. With an increasing deadline, more reconfiguration actions can be scheduled for execution, leading to a higher throughput of reconfiguration actions.

In general, the optimistic mode has approximately 10-15% more remaining reconfiguration actions than the pessimistic mode in all three event arrival rates. This is because of the different scheduling in the two modes, as the pessimistic mode always schedules

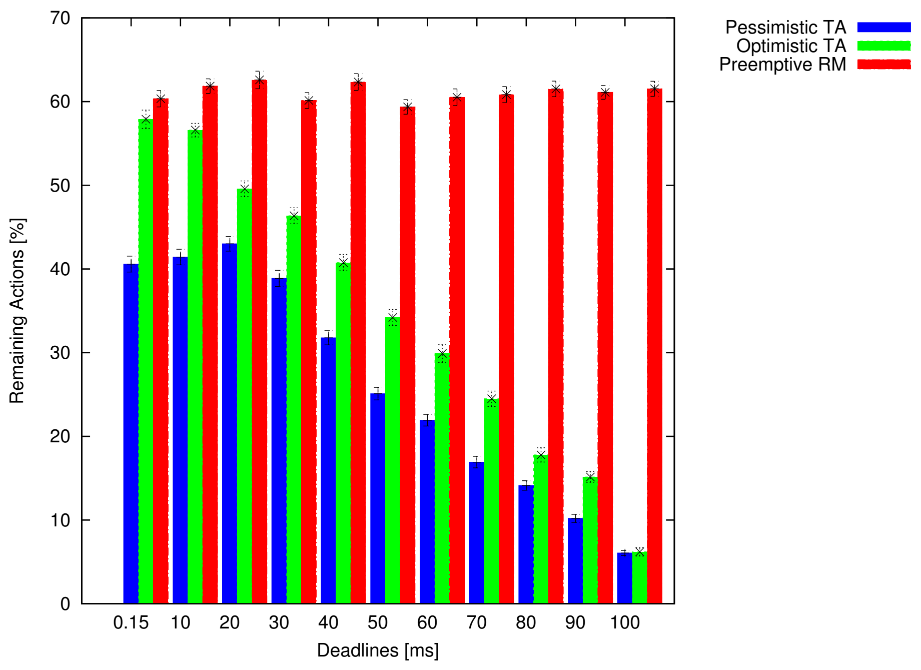


Fig. 5.8: Percentage of remaining actions for medium event arrival rate

complete reconfiguration actions, even if the overall event deadline is missed, whereas the optimistic mode aborts reconfiguration actions in earlier phases. This leads to a higher throughput of reconfiguration actions for the pessimistic mode.

To illustrate the results on a more detailed level, Figures 5.10 and 5.11 illustrate the average number of remaining reconfiguration actions when there are two consecutive events for the pessimistic and optimistic mode, respectively, with a medium event arrival rate. Figures 5.12 and 5.13 show the average number of remaining reconfiguration actions for a very high event arrival rate.

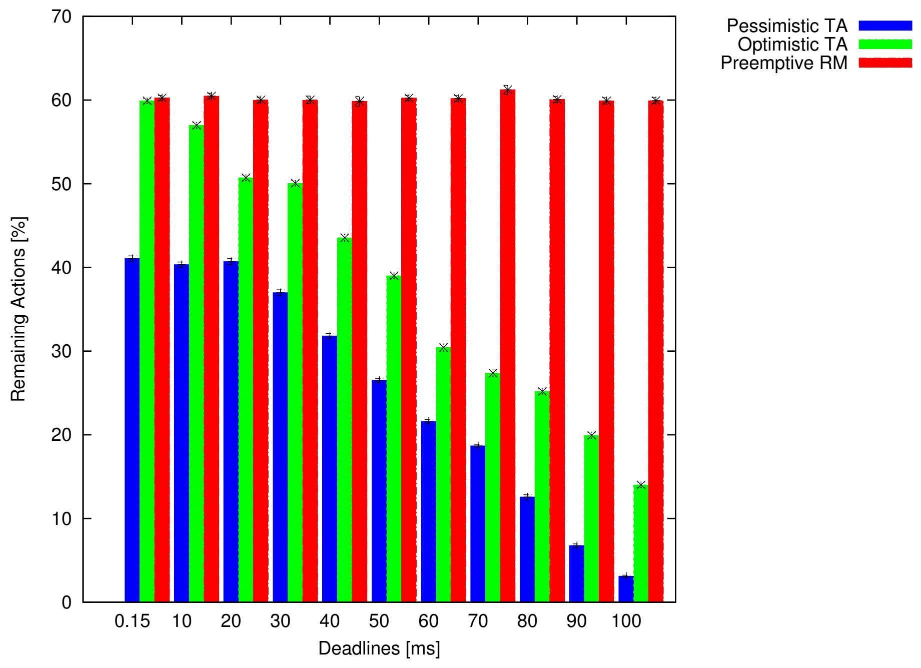


Fig. 5.9: Percentage of remaining actions for high event arrival rate

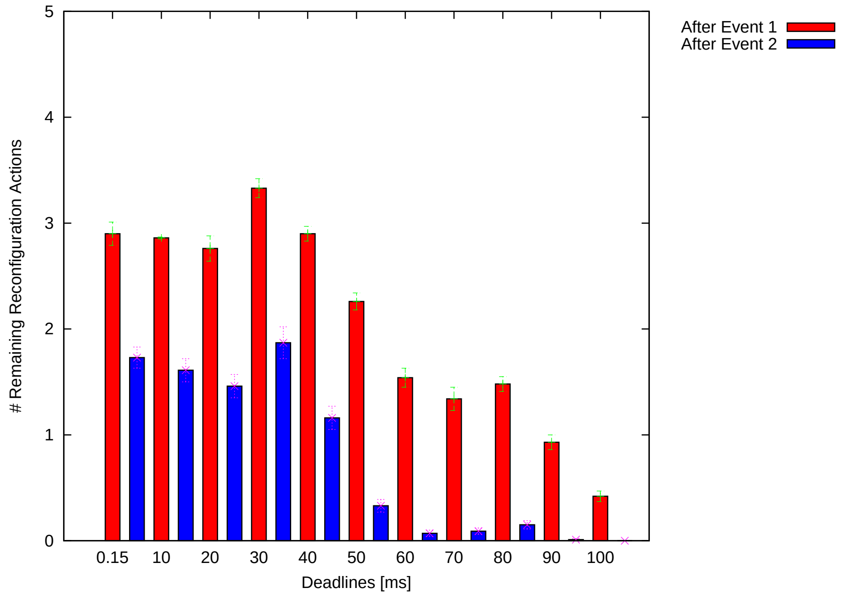


Fig. 5.10: Pessimistic mode: Number of remaining actions for medium event arrival rate and two consecutive events

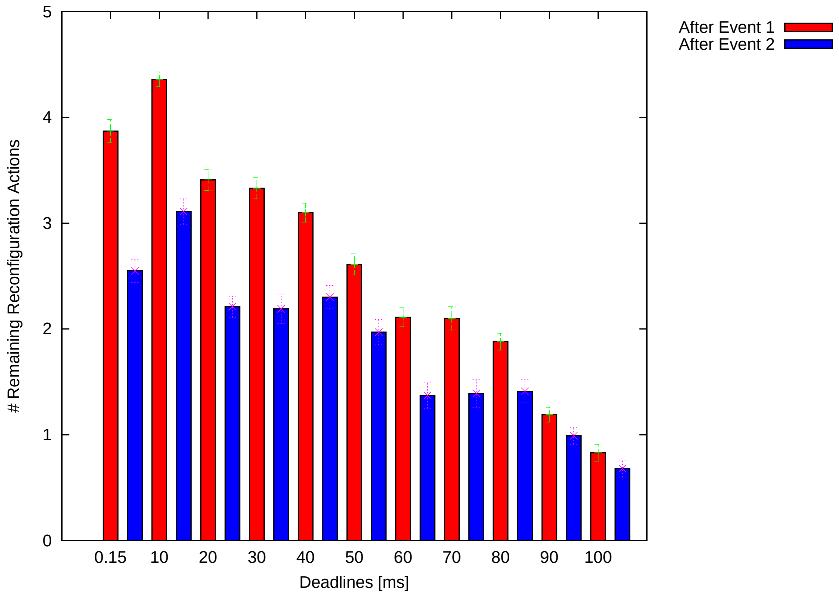


Fig. 5.11: Optimistic mode: Number of remaining actions for medium event arrival rate and two consecutive events

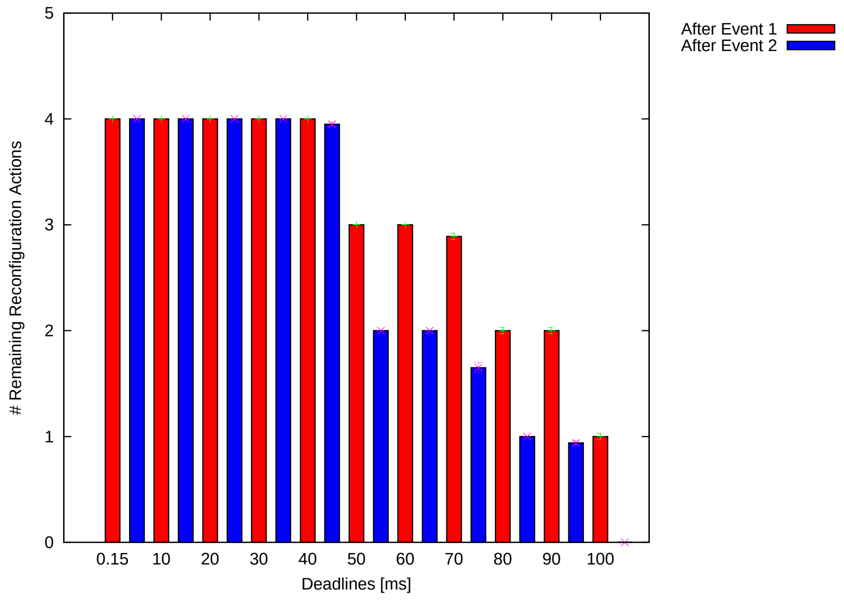


Fig. 5.12: Pessimistic Mode: Number of remaining actions for very high event arrival rate and two consecutive events

In both modes, when the event arrival rate is at a medium rate, the average number of remaining reconfiguration actions varies for two consecutive events (see Figures 5.10, and 5.11). This indicates that some actions have completed. The varying number of remaining reconfiguration actions per event in each mode is caused by the scheduling modes that either execute a complete reconfiguration action (pessimistic mode) or complete or undo the currently executing phase of an active reconfiguration action.

Contrary to the case of medium event arrival rates, the number of remaining reconfiguration actions are the same for two consecutive events in both modes when a high event arrival rate is applied, and deadlines are smaller than 50 ms (see Figures 5.12 and 5.13). The higher number of remaining reconfiguration actions is because, when the event arrival rate is high, there is a higher probability that two consecutive events occur

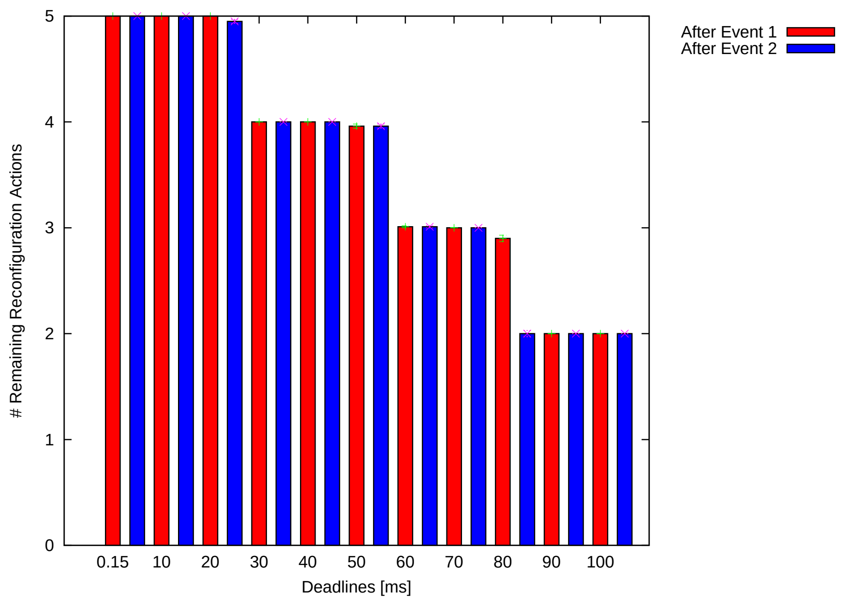


Fig. 5.13: Optimistic Mode: Number of remaining actions for very high event arrival rate and two consecutive events

at the same reconfiguration action. If the processing deadline is smaller than the reconfiguration action execution duration, no additional actions can be scheduled between processing event 1 and processing event 2. This results in the same number of remaining reconfiguration actions for both events and a decreased throughput of reconfiguration actions. For deadlines higher than 50 ms, the pessimistic mode can schedule additional reconfiguration actions when processing event 1 and as a result, the number of remaining reconfiguration actions decreases when processing event 2. In the optimistic mode, the waiting time of event 2 increases with an increasing deadline, as additional reconfiguration actions can be scheduled. However, the increased waiting time leads to a decreased remaining scheduling time until event 2 must be processed. Therefore, event 1 and event 2 have also the same number of remaining reconfiguration actions for higher deadlines.

In summary, both modes schedule more reconfiguration actions than a preemptive reconfiguration model. The higher throughput of reconfiguration actions for various deadlines and event arrival rates fulfils objective O2. However, when the event arrival rate is very high and there are many incoming events during an ongoing reconfiguration, TimeAdapt cannot guarantee a timely completion of the reconfiguration sequence, as it has to wait until the high load has reduced. Embedded software that needs timely completion of a reconfiguration and does not need to react to incoming events, should use a transactional reconfiguration model.

5.3.4 Experiment 2: Heterogeneous Reconfiguration

This experiment evaluates the impact of varying reconfiguration action execution durations on the performance of TimeAdapt with respect to meeting event deadlines and the percentage of remaining reconfiguration actions. Table 5.7 lists the parameter settings of this experiment.

Parameter	Name	Value
Reconfiguration sequence length	l	4,5
Reconfiguration action type	a	upgradeActor, addActor
Event arrival rate [ms]	λ	2000
Event processing deadline [ms]	t_d	0.15...100

Table 5.7: Experiment 2: Parameter Setting

For this experiment a low event arrival rate of 2000 ms is used. This event arrival rate raises a single event over the given reconfiguration sequences and allows the in-depth study of TimeAdapt without additional overhead, such as event queuing times. The two different types of reconfiguration actions used are *addActor* actions and *upgradeActor* actions. *AddActor* actions add temperature filter components to the scenario, whereas *upgradeActor* actions upgrade the existing *SensorActor*, *ControlActor*, and *AlertActor* components. Note that the *connectActor* actions are implicitly contained within the *addActor* reconfiguration actions. In detail, the reconfiguration actions are:

- *upgradeActor(TemperatureSensorActor)*
- *addActor(TemperatureFilterActor)*
- *addActor(TemperatureFilterActor)*
- *upgradeActor(ControlActor)*
- *upgradeActor(AlertActor)*

The execution duration of a single *addActor* reconfiguration action is randomly chosen between 60 ms and 70 ms, which is in the range of the values found by Hughes et al. (2009). The execution duration of an *upgradeActor* reconfiguration action is the same as in experiment 1. The reconfiguration sequences used differ in their execution durations:

- A reconfiguration sequence comprised of five *upgradeActor* actions, having the shortest overall execution duration (*"Upgrade"*) (see Experiment 1).
- A reconfiguration sequence comprised of two *upgradeActor* actions, upgrading the *ControlActor* and *AlertActor* components, as well as two *addActor* actions, adding two *TemperatureFilterActors*. This sequence has an overall medium execution duration(*"Heterogeneous"*).
- A reconfiguration sequence comprised of four *addActor* actions, in which all four *TemperatureFilterActors* are added to the basic configuration. This sequence has the longest overall execution duration (*"Add"*).

The first reconfiguration sequence is composed of non-interface changing reconfiguration actions. The second and third reconfiguration sequences are composed of reconfiguration actions on independent actors. In all three reconfiguration sequences, there is one action in each safe step. The varying execution durations of these sequences result in different target configurations, depending on the size of the processing deadline.

It is expected that TimeAdapt will meet a higher percentage of deadlines for the reconfiguration sequence comprised of short execution duration actions, compared to the reconfiguration sequences comprised of medium and long execution durations, specifically when deadlines are comparatively small. We also expect a lower percentage of reconfiguration actions remaining for the reconfiguration sequence comprised of short execution duration actions, compared to the reconfiguration sequences comprised of medium and long execution durations. Figures 5.14, and 5.15 illustrate the percentage of deadlines met for the pessimistic and optimistic modes and the three reconfiguration sequences applied. Figures 5.16 and 5.17 show the percentage of reconfiguration actions remaining, for both modes.

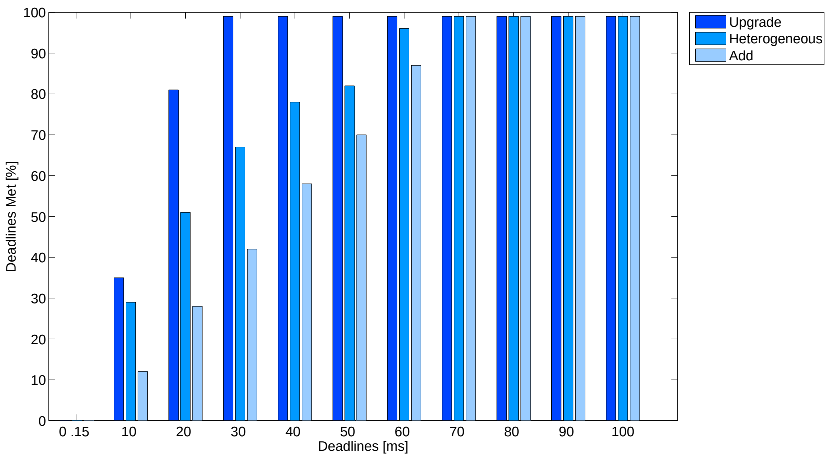


Fig. 5.14: Pessimistic Mode: Percentage of deadlines met for low event arrival rate

In the pessimistic mode, the *upgrade* reconfiguration sequence meets the highest percentage of deadlines when deadlines are smaller or equal than 60 ms. The smallest percentage of deadlines met is reached by the *add* reconfiguration sequence, when deadlines are smaller or equal than 60 ms (see Figure 5.14). These results concur with the

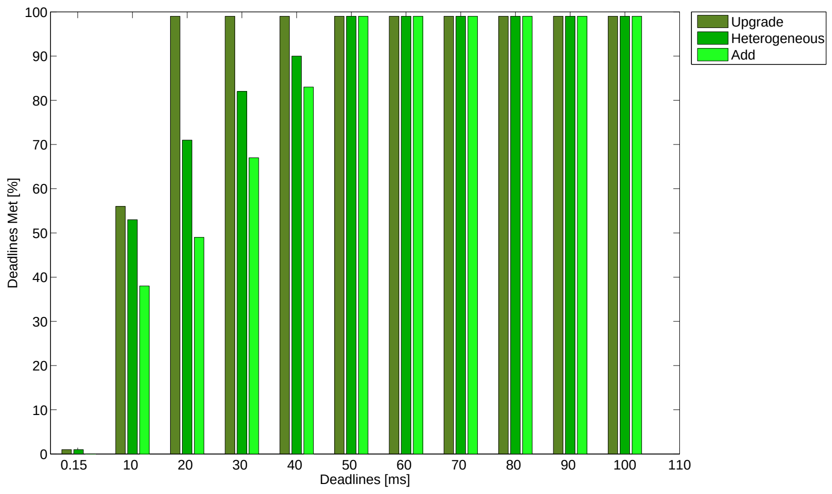


Fig. 5.15: Optimistic Mode: Percentage of deadlines met for low event arrival rate

expectations that shorter reconfiguration sequences will lead to more deadlines met. An *upgradeActor* action takes approximately 25 ms to execute. Incoming events with deadlines greater than 25 ms can be processed within the deadline. Therefore, for deadlines larger than 25 ms, the *upgrade* reconfiguration sequence meets incoming event deadlines. An *addActor* action takes approximately 60 ms and hence only deadlines greater than 60 ms can be met. This is confirmed by the results, as for deadlines higher than 60 ms, all three reconfiguration sequences show the same behaviour.

The optimistic mode shows similar results. For deadlines smaller than 50 ms, the *upgrade* reconfiguration sequence meets the highest percentage of deadlines, whereas the *add* reconfiguration sequence meets the lowest percentage of deadlines (see Figure 5.15). For deadlines equal to or higher than 50 ms, all three reconfiguration sequences show the same behaviour. In general, for deadlines smaller than 50 ms, the optimistic mode meets a higher percentage of deadlines than the pessimistic mode, for all three sequences. The higher percentage can be explained by the finer granularity of reconfiguration scheduling in the optimistic mode, which allows more control as to when to abort a reconfiguration sequence.

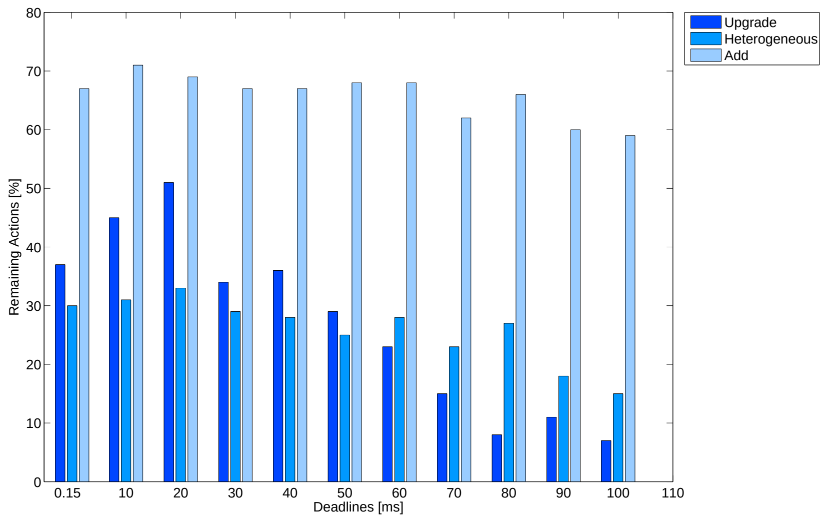


Fig. 5.16: Pessimistic Mode: Percentage of remaining actions for low event arrival rate

Regarding the percentage of reconfiguration actions remaining, the pessimistic mode shows different behaviour for each of the three applied reconfiguration sequences. When using the *upgrade* reconfiguration sequence, the mode has an increasing percentage of actions remaining for deadlines smaller than 30 ms, and a decreasing percentage of actions remaining for deadlines equal to or larger than 30 ms (see Figure 5.16). The slight increase for deadlines smaller than 30 ms is because for deadlines that are smaller than the action's execution duration, the pessimistic mode does not schedule further actions. As a single *upgradeActor* action takes 25 ms, with an increasing deadline more reconfiguration actions can be scheduled, leading to the decrease in the percentage of remaining actions. In contrast, the *heterogeneous* reconfiguration sequence and the *add* reconfiguration sequence show an almost constant percentage of remaining actions for deadlines smaller than 80 ms. For deadlines equal to or higher than 80 ms, both sequences show a decrease in their percentage of remaining reconfiguration actions (see Figure 5.16). The constant number of remaining reconfiguration actions for deadlines smaller than

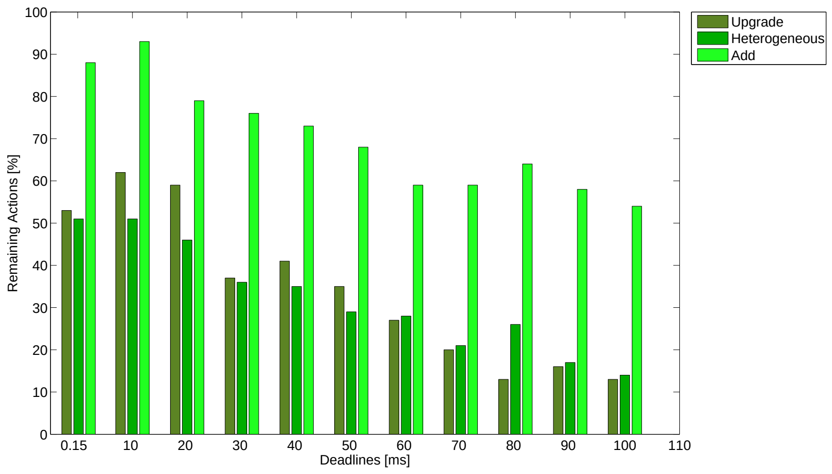


Fig. 5.17: Optimistic Mode: Percentage of remaining actions for low event arrival rate

80 ms can be explained by the larger execution durations of single actions. When the deadline is smaller than the actual reconfiguration action execution duration, the pessimistic mode aborts the scheduling of actions after completion of the current action. When deadlines are higher, more reconfiguration actions can be scheduled within this deadline, leading to a higher throughput of reconfiguration actions.

The optimistic mode shows the same behaviour as the pessimistic mode for the *upgrade* reconfiguration sequence. There is a slight increase in the percentage of remaining reconfiguration actions for deadlines smaller than 30 ms, and for deadlines equal to or larger than 30 ms a decreasing percentage of remaining reconfiguration actions (see Figure 5.17). Like in the pessimistic mode, the slight increase can be explained by the fact that for deadlines that are smaller than an *upgradeActor* action, the optimistic mode does not schedule further actions. With an increasing deadline, more reconfiguration actions can be scheduled, resulting in a higher throughput. In contrast to the pessimistic mode, in the optimistic mode, the percentage of remaining reconfiguration sequences for the *heterogeneous* and *add* reconfiguration sequence decreases when the deadline increases (see Figure 5.17). The decrease is due to the more fine-grained scheduling mode.

However, this leads to a scheduling overhead, which results in approximately 10% more remaining reconfiguration actions per sequence for the optimistic mode compared to the pessimistic mode.

It can be concluded from this experiment that the percentage of deadlines met and the percentage of remaining reconfiguration actions is dependent on the execution durations of the reconfiguration actions in a reconfiguration sequence. A reconfiguration sequence composed of reconfiguration actions that have a higher execution duration needs a higher deadline to guarantee that events can be processed within time. In this experiment, the homogeneous reconfiguration sequence comprised of *addActor* reconfiguration actions has the longest overall execution duration, and as a result meets less deadlines than the heterogeneous reconfiguration sequence or the reconfiguration sequence comprised of *upgradeActor* actions. This sequence also has the highest percentage of remaining reconfiguration actions, as because of its long execution duration, not all actions can be executed within a given deadline. TimeAdapt's optimistic mode has a higher percentage of remaining reconfiguration actions than the pessimistic mode, independent of an action's execution duration, as it has a higher scheduling time and reaches faster an event's deadline. However, the percentage of deadlines met and remaining reconfiguration actions in this experiment are comparable to the results of experiment 1. Hence, performance objective O1 and O2 are fulfilled.

5.3.5 Experiment 3: Varying Safe Step Size

This experiment evaluates the impact of safe step execution durations on the percentage of deadlines met and the percentage of reconfiguration actions remaining. In this experiment, only the pessimistic mode is considered, as the optimistic mode does not consider safe steps, but rather single reconfiguration actions. Table 5.8 lists the parameter settings of this experiment.

For this experiment we used three reconfiguration sequences with varying number of reconfiguration actions and a resulting varying execution duration:

Parameter	Name	Value
Reconfiguration sequence length	l	4,5
Reconfiguration action type	a	upgradeActor, addActor, replaceActor
Event arrival rate [ms]	λ	2000
Event processing deadline [ms]	t_d	0.15...100

Table 5.8: Experiment 3: Parameter Setting

- *Sequence 1:* A reconfiguration sequence comprised of five *upgradeActor* actions, each of which upgrades a component in the scenario. In detail, the reconfiguration actions are:
  - *upgradeActor(tempSensor)*
  - *upgradeActor(Control)*
  - *upgradeActor(Filter)*
  - *upgradeActor(Alert)*
  - *upgradeActor(Output)*

  As *upgradeActor* actions are non-interface changing, each actions is partitioned into its ows safe step. For this experiment setup, single action safe steps have the shortest execution duration.

- *Sequence 2:* A reconfiguration sequence comprised of two *addActor* and two *connectActor* reconfiguration actions. Note that the *addActor* reconfiguration actions implicitly contain *connectActor* actions. In detail, the reconfiguration actions are:
  - *addActor(TemperatureFilterActor1)*
  - *connectActor(TemperatureSensorActor,TemperatureFilterActor)*
  - *addActor(TemperatureFilterActor2)*
  - *connectActor(TemperatureFilterActor1,TemperatureFilterActor2)*

This results in two safe steps, containing each an *addActor* action and a *connectActor* action. For this experiment setup, these safe steps have a medium execution duration.

- *Sequence 3:* A reconfiguration sequence comprised of four *replaceActor* actions, each of which replaces a component in the scenario with a new component with changed input and output sets. In detail, the reconfiguration actions are:
  - *replaceActor(Control)*
  - *replaceActor(Filter)*
  - *replaceActor(Alert)*
  - *replaceActor(Output)*

  As all of these actions are interface-changing, and address dependent actors, they are grouped together in a single safe step. In this experiment setup, this safe step has the longest execution duration.

As in experiment 2, a low event arrival rate is used to investigate TimeAdapt's behaviour in more detail.

As shown in Figure 5.18, for deadlines greater than 20 ms, the reconfiguration sequence with the shortest overall execution duration of its safe steps meets the highest percentage of deadlines (*Sequence1*). A safe configuration is reached after the execution of a single reconfiguration action. As this reconfiguration sequence is comprised of single *upgradeActor* reconfiguration actions, each with an execution duration of 25 ms, this is the case for deadlines larger than 25 ms. The worst behaviour with respect to meeting deadlines is shown by the reconfiguration sequence comprised of safe steps with four actions (*Sequence 3*). In this case, a safe configuration is reached only when all actions are executed. The percentage of deadlines met depends on the time the event occurs and its associated deadline. Larger deadlines have a greater probability of fitting the remaining reconfiguration sequence's execution duration. A reconfiguration sequence with safe

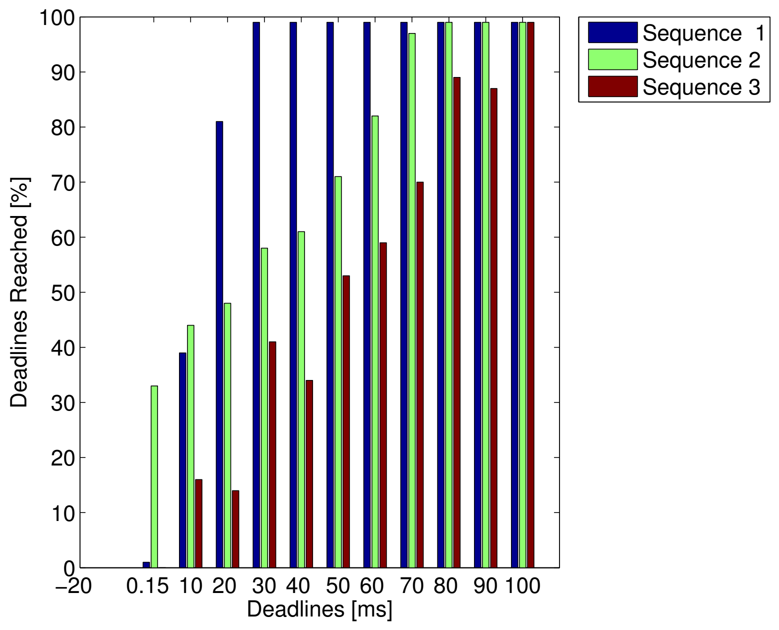


Fig. 5.18: Percentage of deadlines met for low event arrival rate

steps containing two actions (*Sequence 2*) meets a higher percentage of deadlines than a reconfiguration sequence with safe steps containing four actions, as only two reconfiguration actions need to be executed to reach a safe system configuration. Nevertheless, this sequence meets a smaller percentage of deadlines than the sequence comprised of single safe steps.

Figure 5.19 illustrates the percentage of remaining reconfiguration actions for the three different safe step sizes. When the safe step has four reconfiguration actions (*Sequence 3*), the percentage for each deadline is always zero, i.e., there are no remaining reconfiguration actions. This is because all actions in the reconfiguration sequence are in this safe step and the pessimistic mode therefore behaves like a transactional reconfiguration model in this case.

For deadlines smaller than 60 ms, the percentage of remaining reconfiguration actions is lower for the reconfiguration sequence comprised of two actions per safe step

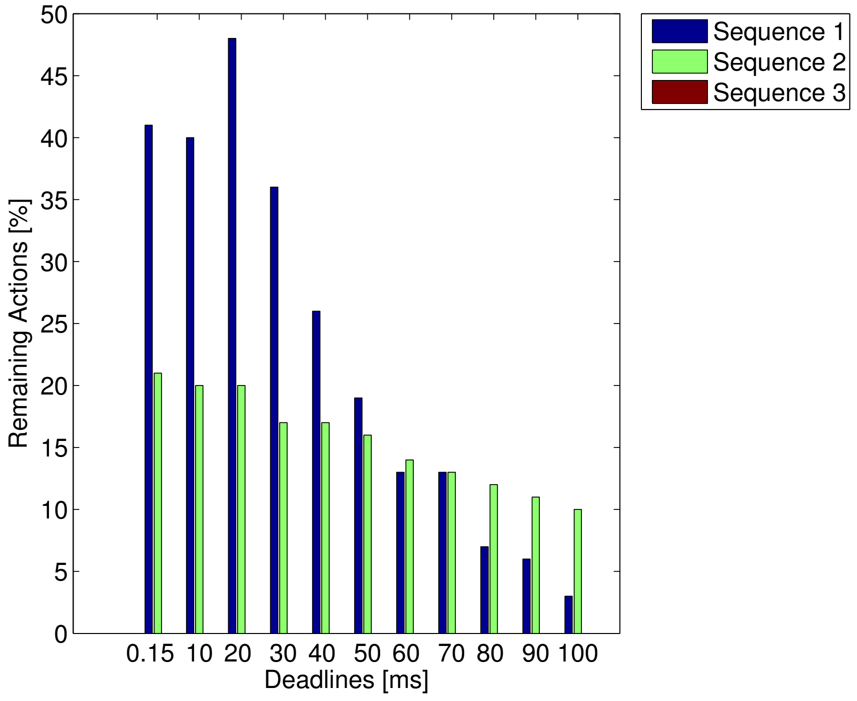


Fig. 5.19: Percentage of remaining actions for low event arrival rate

(*Sequence2*), see Figure 5.19. The larger percentage of remaining reconfiguration actions for single-action safe steps (*Sequence1*) is a result of the varying execution duration of the reconfiguration actions. The single-action safe step comprises *upgradeActor* actions with an execution duration of 25 ms, whereas the sequence containing larger safe steps contains *addActor* actions with an execution duration of 60 ms. The scheduling mode always completes a currently executing safe step before an incoming event can be processed, independently of the deadline. The reconfiguration sequence comprised of single-action safe steps, can be paused much earlier to process an incoming event than for the reconfiguration sequence comprised of multiple actions in the safe step, as it has a smaller execution duration. As an effect, the percentage of remaining reconfiguration actions is higher for single-action safe steps. As the deadline increases, additional reconfiguration actions can be scheduled for the single-action safe steps. In this experiment for deadlines larger than 60 ms, the percentage of remaining reconfiguration actions is less for single-action safe steps than for safe steps comprised of multiple reconfiguration actions. However, the parameter settings were chosen in such a way, that single-action safe steps have a smaller execution duration than safe steps that consists of multiple actions.

We can conclude from these results that the execution durations of safe steps has an impact on TimeAdapt's timeliness as well as its throughput. The higher the execution duration of a safe step, the higher the deadline needs to be so that it can be met. The actual size of a safe step has an impact on the percentage of remaining reconfiguration actions. The more reconfiguration actions need to be executed atomically, the higher the given throughput of reconfiguration actions. In one case, the complete reconfiguration sequence is contained in a single safe step, which leads to a transactional behaviour with respect to remaining reconfiguration actions and percentage of deadlines met. Overall, the percentage of deadlines met is still higher for safe steps with a smaller execution duration compared to a transactional reconfiguration model. Also, the percentage of remaining reconfiguration actions is lower when compared to a preemptive reconfiguration model. Hence, this experiment confirms performance objectives O1 and O2.

5.3.6 Experiment 4: Multiple Event Sources

This experiment evaluates the percentage of deadlines met and the percentage of reconfiguration actions remaining, when multiple event sources emit events. Table 5.9 summarises the parameter values used.

Events are emitted using a medium event arrival rate. This event arrival rate ensures that there are queued events. Event deadlines are the same as used in experiment 1. The same reconfiguration sequence as in Experiment 1 is used:

- *upgradeActor(tempSensor)*

Parameter	Name	Value
Reconfiguration sequence length	l	5
Reconfiguration action type	a	upgradeActor
Event arrival rate [ms]	λ	35.0
Event processing deadline [ms]	t_d	0.15. . .100

Table 5.9: Experiment 4: Parameter Setting

- *upgradeActor(Control)*
- *upgradeActor(Filter)*
- *upgradeActor(Alert)*
- *upgradeActor(Output)*

This implies that there are five safe steps, each containing a single *upgradeActor* action. The experiment itself is divided into two parts: In the first part, events of both sources have the same associated deadline in each experiment run. In the second part, events of both sources have different deadlines.

5.3.6.1 Homogeneous Event Deadlines

Figures 5.20 and 5.21 illustrate the percentage of deadlines met for TimeAdapt's pessimistic and optimistic modes when events are generated by two event sources and have the same associated deadlines.

In the pessimistic mode, the deadline of events generated by event source 1 is met in approximately 3% more cases than the deadline of events generated by event source 2 for deadlines smaller than 30 ms (see Figure 5.20). For deadlines smaller than 10 ms this gap is even wider, with 12% of deadlines met for events generated by event source 1 and 0% of deadlines met for events generated by event source 2. The difference is because events from both sources occur at the same time on the same reconfiguration

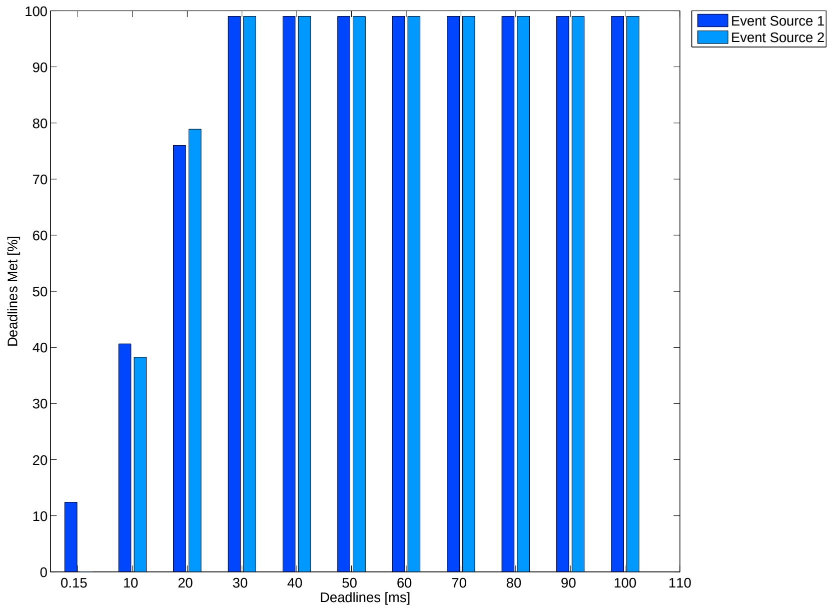


Fig. 5.20: Pessimistic Mode: Percentage of deadlines met for medium event arrival rate and homogeneous deadlines

action. At each point in time, TimeAdapt processes only a single event. Events that occur simultaneously are queued and will be processed after a delay. Queueing the event, however, leads to a higher waiting time and so, under the same deadline, it is more likely that at least one of the event deadlines will be missed. When the deadline is large enough (i.e., over 30 ms), both event sources have the same percentage of their deadlines met. These findings are consistent with the findings for a single event source and a medium event arrival rate (see Figure 5.3).

The optimistic mode shows a similar difference in the percentage of deadlines met for events generated by event source 1 and event source 2 (see Figure 5.21). Two differences to the pessimistic mode can be found for very small deadlines, i.e., 0.15 ms, and very large deadlines, such as 100 ms. When the deadline is very small, the percentage of deadlines met is the same for both event sources, namely 46%. This higher percentage

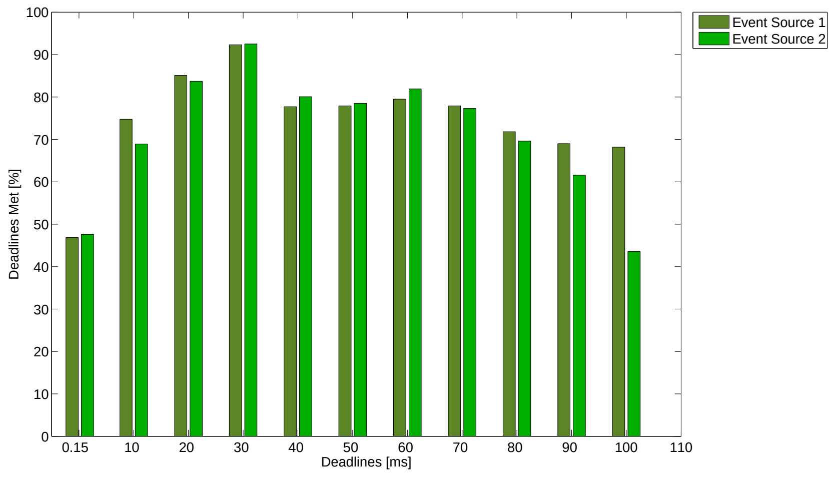


Fig. 5.21: Optimistic Mode: Percentage of deadlines met for medium event arrival rate and homogeneous deadlines

is because of the more fine-grained scheduling that allows higher control over when a reconfiguration sequence can be aborted. In the case of large deadlines, such as 100 ms, the optimistic mode meets a smaller percentage of deadlines for events emitted by event source 2, compared to smaller deadlines. The decreased percentage is because of the more complex scheduling by the optimistic mode, which leads to an increased waiting time for queued events. This is true when events are either from the same or from a different event source. Note that due to the additional overhead of queued events, the percentage of deadlines met is approximately 10-15% lower as the results when a single event source and medium event arrival rate is applied (see Figure 5.3).

We can conclude from these results that the number of event sources has only a little impact on the percentage of deadlines met in both modes when event sources have homogeneous deadlines.

Figures 5.22 (pessimistic mode) and 5.23 (optimistic mode) show the percentage of remaining reconfiguration actions, when events are emitted from two event sources. In both modes, and for both event sources, the percentage of remaining reconfiguration actions decreases with an increasing deadline.

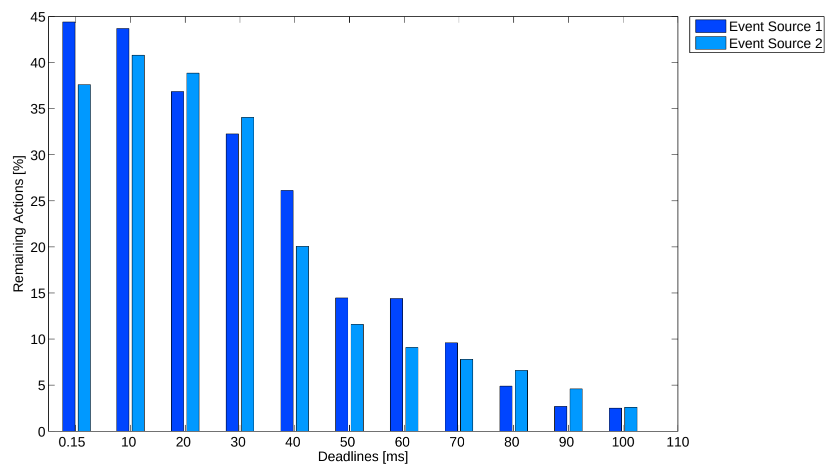


Fig. 5.22: Pessimistic Mode: Percentage of remaining actions for medium event arrival rate and homogeneous deadlines

In the pessimistic mode, either events generated by event source 1 or events generated by event source 2 have a higher percentage of remaining reconfiguration actions (see Figure 5.22). This is because only one event can be processed at each point in time and as a result at least a single reconfiguration action was completed. It is non-deterministic as to which event is scheduled, which accounts for the variance.

The same effect with respect to variations in the percentage of remaining actions for events generated by event source 1 and event source 2 can be found in the optimistic mode (see Figure 5.23). Like in the pessimistic mode, the variation is because events from both event sources occur at the same time, but need to be processed sequentially.

When there are multiple event sources, all with the same arrival rate, the number of events queued is higher than when a single event source is applied. The waiting time of these queued events needs to be deducted from the deadline. Hence, the throughput of actions is less when multiple event sources are applied than when a single event source is applied, under the same given deadline. Note that the percentage of remaining actions

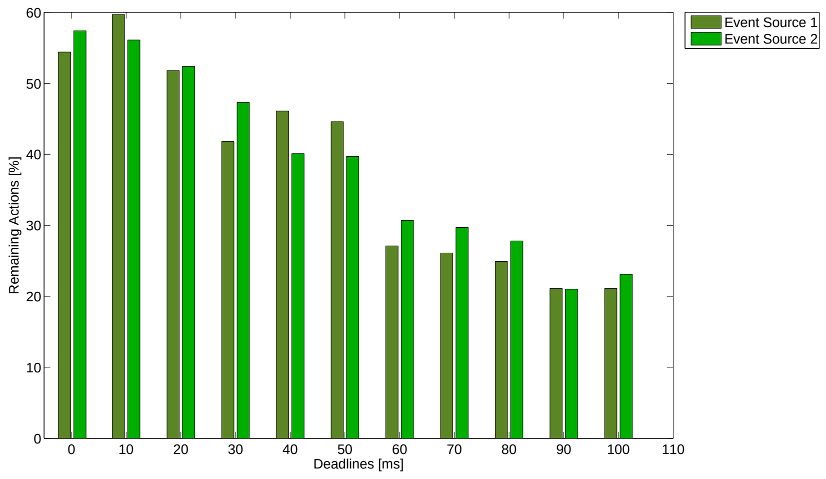


Fig. 5.23: Optimistic Mode: Percentage of remaining actions for medium event arrival rate and homogeneous deadlines

for multiple event sources is roughly the same for both modes of TimeAdapt as for single event sources under a higher event arrival rate, such as 8 ms, which confirms that a higher number of queued events impacts reconfiguration action throughput rate (see Figures 5.12 and 5.13).

5.3.6.2 Heterogeneous Event Deadlines

This experiment compares the percentage of deadlines met and the percentage of remaining reconfiguration actions, when using multiple event sources with varying associated deadlines for their generated events. For this experiment, deadlines for events from event source 2 always are twice as large as deadlines associated with events from event source 1. The shorter deadline of events from event source 1 results in queued events from this source to be processed before queued events from event source 2. Table 5.10 summarises the deadline values used:

Figures 5.24 and 5.25 show the percentage of deadlines met for TimeAdapt's pessimistic mode and optimistic mode, respectively.

Event Source 1 [ms]	Event Source 2 [ms]
1	2
2.5	5
5	10
10	20
20	40
40	80
50	100

Table 5.10: Deadline values associated with events

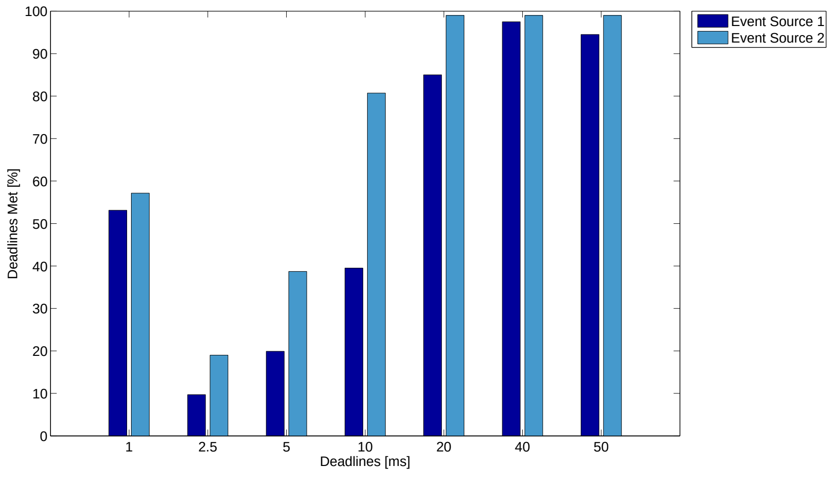


Fig. 5.24: Pessimistic Mode: Percentage of deadlines met for medium event arrival rate and heterogeneous deadlines

In the pessimistic mode, events generated by event source 2 have, in all cases, a higher percentage of their deadlines met than events generated by event source 1 (see Figure 5.24). This can be explained by the fact that even if these events are queued and have an associated waiting time, the higher deadline leads to a higher chance that their associated deadline will be met. An interesting observation in this experiment is that

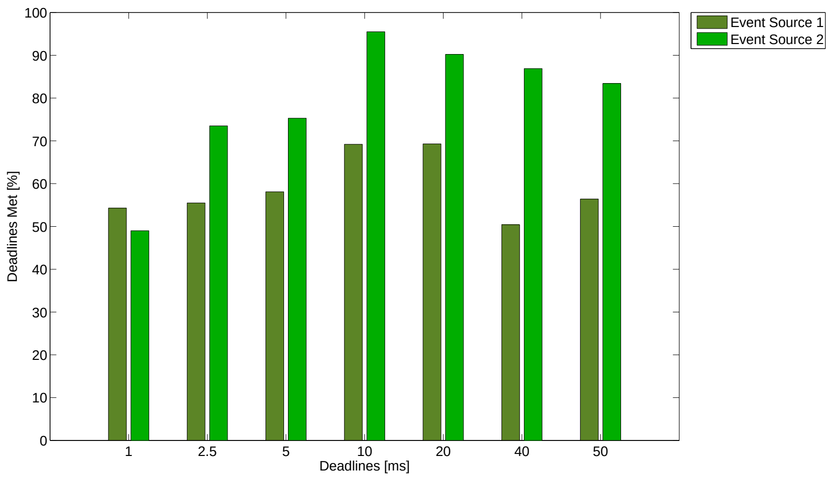


Fig. 5.25: Optimistic Mode: Percentage of deadlines met for medium event arrival rate and heterogeneous deadlines

events that are generated by event source 1 have, in general, a lower percentage of their deadlines met compared to their respective counterparts, where both event sources have the same associated deadline (see Figure 5.20). If an event from event source 2 happens to occur before an event from event source 1, the pessimistic mode can schedule more reconfiguration actions for execution because of their higher deadline. This increases the waiting time for events in the queue. The higher waiting time leads to a higher likelihood of missing the associated deadline.

Figure 5.25 shows the percentage of deadlines met for the optimistic mode. Like in the pessimistic mode, events generated by event source 2 have, in all cases, a higher percentage of their deadlines met than events generated by event source 1. Again, this is the result of higher associated deadlines for events generated by event source 2. Likewise, events generated by event source 1 have a lower percentage of their deadlines met than their respective counterparts, where both event sources have the same associated deadline (see Figure 5.25). Similar to the pessimistic mode, this is because of the higher deadline associated with events from event source 2 and the resulting higher waiting time for events generated by event source 1.

Figures 5.26 and 5.27 show the percentage of remaining reconfiguration actions for the pessimistic and optimistic mode, respectively.

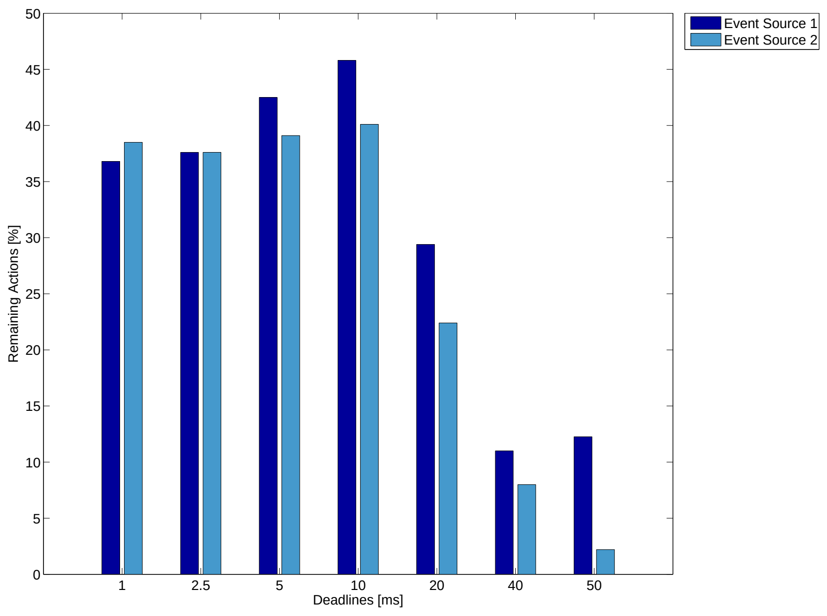


Fig. 5.26: Pessimistic Mode: Percentage of remaining actions for medium event arrival rate and heterogeneous deadlines

In the pessimistic mode, the percentage of remaining actions depends on whether the currently processed event is from event source 1 or event source 2. In general, for all deadlines, events from event source 2 have a smaller percentage of remaining reconfiguration actions than events from event source 1. If an event from event source 1 can be directly processed, the small deadline leads only to the completion of the current reconfiguration action. If an event from event source 1 occurs, while an event from event source 2 is being processed by TimeAdapt, this event is queued. However, the higher associated deadlines of events from event source 2 lead to a higher queuing time, and as a result TimeAdapt might not schedule further actions when processing an event from

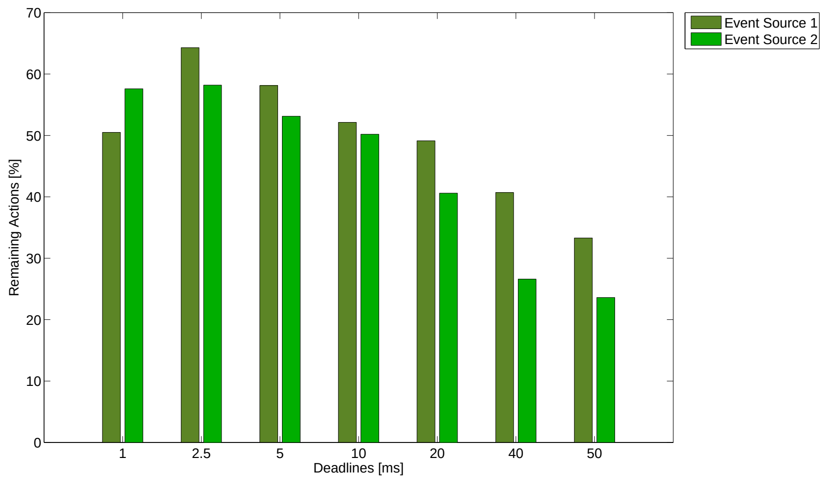


Fig. 5.27: Optimistic Mode: Percentage of remaining actions for medium event arrival rate and heterogeneous deadlines

event source 1. For deadlines smaller than 20 ms, the percentage of remaining actions slightly increases for events generated by event source 1 (see Figure 5.26). The slight increase is because of the increased waiting time for events generated by event source 1. With a higher associated deadline, the percentage of remaining reconfiguration actions decreases. For deadlines smaller than 20 ms, the percentage of remaining actions remains approximately constant at 38% for events generated by event source 2. This is because the reconfiguration action times are between 15 and 25 ms. A deadline of 20 ms allows only the scheduling of a single reconfiguration action. With an increased deadline size, the percentage of remaining actions also decreases, as more actions can be executed.

The effect of heterogeneous deadlines is confirmed by the results for the optimistic mode (see Figure 5.27). Like in the pessimistic mode, an increasing deadline leads to a decrease in the number of remaining reconfiguration actions, as more reconfiguration actions can be executed.

We can conclude from all the obtained results that the impact of multiple event sources on the timeliness and throughput behaviour of TimeAdapt depends on the number of event sources and whether events of different event sources have the same or different associated deadlines. A higher number of event sources results in a higher number of potentially queued events with an increased waiting time. The effect of queued events on timeliness and throughput depends on whether the events have homogeneous or heterogeneous deadlines. Homogeneous deadlines affect the throughput of a reconfiguration, especially when the associated deadline of the queued event is high enough to fit the waiting time but too small to schedule the next reconfiguration action. In this case, the deadline of the queued event is met, however, no progress with an ongoing reconfiguration can be made. Heterogeneous deadlines affect the timeliness of a reconfiguration, especially when the queued event has an associated small deadline, as then the deadline is more likely to be missed. In general, both modes of TimeAdapt have a higher number of deadlines met than a transactional reconfiguration model and have lower number of remaining reconfiguration actions than a preemptive reconfiguration model. These results are comparable to results of experiment 1, fulfilling objectives O1 and O2. However, as already discussed in Section 5.2, embedded software that needs strict guarantees on event deadlines, should use a preemptive reconfiguration model, whereas embedded software that needs a timely completion of a reconfiguration should apply a transactional reconfiguration model.

5.3.7 Experiment 5: Reconfiguration Execution Overhead

This experiment evaluates TimeAdapt's inherent overhead in terms of reconfiguration execution duration by measuring the time elapsed from the start of a reconfiguration until a reconfiguration is completed. Table 5.11 summarises the parameter values used. The same settings with respect to the length and the type of reconfiguration action were used as in experiment 1, 3, and 4, i.e., a reconfiguration sequence is comprised of five *upgradeActor* reconfiguration actions. In detail, the reconfiguration actions are:

- *upgradeActor(tempSensor)*

Parameter	Name	Value
Reconfiguration sequence length	l	5
Reconfiguration action type	a	upgradeActor
Event arrival rate [ms]	λ	35.0, 8.0
Event processing deadline [ms]	t_d	0.15...100

Table 5.11: Experiment 5: Parameter Setting

- *upgradeActor(Control)*
- *upgradeActor(Filter)*
- *upgradeActor(Alert)*
- *upgradeActor(Output)*

The experiment is executed in three different scenarios, which differ in the event arrival rate used and the deadline associated.

1. Scenario 1: There are no incoming events. This scenario illustrates the baseline for reconfiguration execution duration overhead as TimeAdapt is not being used.
2. Scenario 2: This scenario uses a medium event arrival rate and a deadline that is smaller than the execution time of a single reconfiguration action, (10 ms). Under these conditions, there is no further scheduling of reconfiguration actions.
3. Scenario 3: This scenario uses a high event arrival rate and the same deadline as in Scenario 2. Due to the high event arrival rate, events are queued and the additional TimeAdapt overhead is added to each reconfiguration action execution duration.

Figure 5.28 shows the results for the pessimistic mode, and Figure 5.29 shows the results for the optimistic mode.

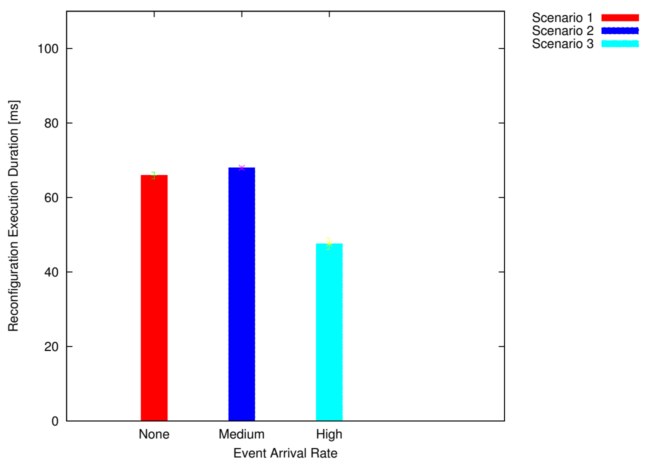


Fig. 5.28: Total execution duration for pessimistic mode

The execution duration overhead for the pessimistic mode is approximately 2% for scenario 2, which comprises a medium event arrival rate and a small deadline, compared to the baseline, i.e., scenario 1. This increase in execution duration is because of the additional scheduling phase that happens when an event occurs. Scenario 3 has an increased overhead of 5%, compared to Scenario 1, and of 2% compared to Scenario 2. The additional overhead is because of a higher number of queued events that need to be scheduled. However, as the deadline is too small to schedule any reconfiguration action for execution, events are directly processed, resulting only in a small overhead compared to the execution durations of Scenario 1 and Scenario 2.

A different result is obtained for the optimistic mode. In this mode, the lowest execution duration is also achieved by scenario 1, i.e., the reconfiguration sequence in which no event occurs (see Figure 5.29). In contrast to the pessimistic mode, scenario 3 has an execution duration overhead of approximately 2%, compared to scenario 1. This overhead can be explained by the higher scheduling costs, compared to the pessimistic mode. The highest execution duration is shown by scenario 2, which has an approximately 30% higher execution duration than the other two scenarios. The high execution duration is the result of the more complex scheduling mode performed by the optimistic approach.

Note that this scenario has a higher execution duration than scenario 3, where the event arrival rate is higher. This is because queued events are not scheduled as their waiting time exceeds their deadline.

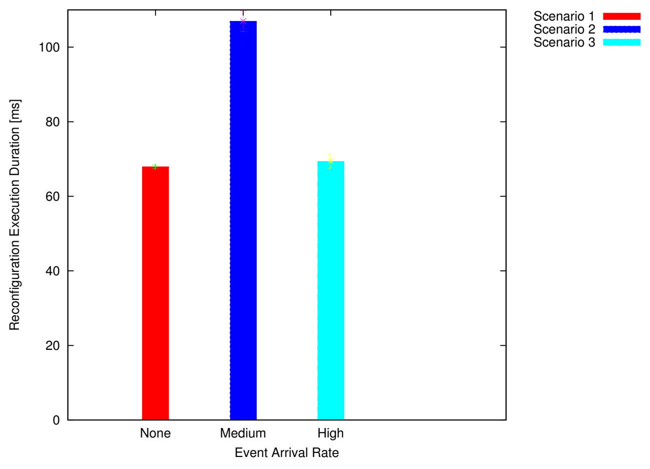


Fig. 5.29: Total execution duration for optimistic mode

In summary, in both modes, the execution duration increases with an increased event arrival rate. The occurrence of events imposes execution duration overhead compared to when no events occur. A lower event arrival rate has a lower execution duration compared to a higher event arrival rate, as queued events impose additional overhead. For the given scenarios, the worst-case overhead for the pessimistic mode was 6%, fulfilling objective O3. As the optimistic mode applies a more complex scheduling scheme, the reconfiguration execution duration overhead can be over 30%, which is significantly higher than when no events are considered. A trade-off must be made between a fast execution of a given reconfiguration, achieved by the pessimistic mode, versus a timely reaction to incoming events, achieved by the optimistic mode.

5.4 Summary

This chapter outlines a set of experiments that assess the underlying tradeoff within TimeAdapt between meeting more deadlines of incoming events than a transactional reconfiguration model (Objective O1), and making more progress towards reconfiguration completion than a preemptive reconfiguration model (Objective O2). TimeAdapt outperforms a baseline implementation of a transactional reconfiguration model with respect to the percentage of deadlines met. The improvement ranges from 31% to 90% for the pessimistic mode, and from 1% to 82% for the optimistic mode, depending on the deadline and the event arrival rate. TimeAdapt also outperforms a baseline implementation of a preemptive reconfiguration model with respect to the throughput of reconfiguration actions, i.e., the percentage of actions remaining. The improvement ranges from 20% to 60% for the pessimistic mode, and from 4% to 53% for the optimistic mode, depending on the deadline and the event arrival rate. This fulfills objectives O1 and O2. However, there is a tradeoff between meeting each event deadline and making progress with a reconfiguration. As discussed in Chapter 3, Section 3.4.2.4, TimeAdapt cannot guarantee the meeting of each event's deadline. Embedded software that needs strong guarantees on the meeting of its event deadlines should use a preemptive reconfiguration model. Similarly, in cases when there are many incoming events during an ongoing reconfiguration, TimeAdapt cannot guarantee the progress towards reconfiguration completion. Embedded software that needs a timely reconfiguration completion, and which does not need to react to incoming events, should use a transactional reconfiguration model.

The experiments also illustrate the impact of a reconfiguration action's execution duration on TimeAdapt's percentage of deadlines met and reconfiguration actions remaining. A reconfiguration sequence comprised of actions with a short execution duration, such as *upgradeActor* actions meets a higher percentage of deadlines, compared to sequences that are comprised of reconfiguration actions with a high execution duration, such as *addActor* actions. This improvement is in the range of 1% to 53% for the pessimistic mode, and 1% to 50% for the optimistic mode, depending on the associated deadline. Also, a reconfiguration sequence comprised of actions with a short execution duration can execute more actions within a deadline than a sequence comprised of actions with a high execution duration. The improvement with respect to reconfiguration actions remaining is in the range of 18% to 52% for the pessimistic mode, and 20% to 41% for the optimistic mode, depending on the associated deadline.

The execution duration of reconfiguration actions particularly influences TimeAdapt's pessimistic mode with respect to the overall execution duration of a safe step. As TimeAdapt's optimistic mode considers only single reconfiguration actions, it is not considered in this experiment. A reconfiguration sequence comprised of safe steps with an overall small execution duration meets a higher percentage of deadlines than a reconfiguration sequence comprised of safe steps with a high execution duration. In the experiment, this resulted in 1% to 55% higher percentage of deadlines met, depending on the size of a deadline. This is because safe steps need to be executed atomically and only after all actions, contained in a safe step, are executed, an incoming event can be processed. The percentage of reconfiguration actions remaining is influenced by the actual size, i.e., number of reconfiguration actions of a safe step. As safe steps are always executed atomically, the more reconfiguration actions are contained in a single safe step, the less reconfiguration actions are remaining, when an incoming event is processed.

Multiple event sources also impact the behaviour of TimeAdapt. This is because the number of queued events is higher when multiple event sources are applied, and the waiting time of the queued events needs to be deducted from the available deadline. The extent of this effect depends on whether the deadlines of the emitted events are homogeneous or heterogeneous. When deadlines are homogeneous, the percentage of met deadlines is approximately 10-15% lower as for the findings when a single event source is applied. The percentage of reconfiguration actions remaining is 0.1-5% higher compared to when a single event source is applied. When deadlines are heterogeneous, the deadlines of queued events is easier missed, especially when the queued event has an associated small deadline. However, when the currently processed event has a large enough deadline, more reconfiguration actions can be scheduled, leading to a higher throughput, compared to homogeneous deadlines.

Finally, both modes of TimeAdapt have an overhead on the overall execution duration of a reconfiguration sequence. This overhead is, in the worst case, approximately 30% compared to the execution duration of a sequence with no incoming event. However, the biggest overhead occurs only when extreme operational settings are applied, such as very small deadlines and a high event arrival rate. With a decreasing event arrival rate, the execution duration overhead reduces to 2%, fulfilling the objective of a small overhead (Objective O3).

From this discussion, we conclude that TimeAdapt is a suitable representation of a time-adaptive reconfiguration model for embedded software that needs to react to incoming events during its reconfiguration, but that can also deal with potential event losses. Examples for such software are signal processing applications or sensing applications in sensor nodes. As the evaluation was performed for platform-specific values for event deadlines and event arrival rates, it needs to be verified that results are similar for other embedded platforms and their respective parameter values.

Chapter 6

Conclusion and Future Work

This thesis describes the design and implementation of TimeAdapt, a novel reconfiguration model for embedded software. TimeAdapt realises a time-adaptive execution model by providing mechanisms that support the timely reaction to incoming events during an ongoing reconfiguration, while progress towards reconfiguration completion is made. This chapter summarises the achievements of this thesis and its contributions, and concludes with a discussion of potential areas of future work.

6.1 Achievements

Embedded software that is executed on a reactive embedded system often requires changes to its software structure when, for example, there is new functionality available or environmental conditions change. As reactive embedded systems have high reliability and durability constraints, these changes need to be executed, while the software is running, without stopping the system. However, embedded systems impose additional challenges on any processes that dynamically change their software, because these systems need to react to incoming events within their associated processing deadlines. At the same time, reconfigurations should be completed in a manner as timely as possible. An analysis of state of the art reconfiguration models that target various kinds of embedded software, such as embedded operating software, or adaptive middleware software, highlighted the two main limitations that motivated the work presented in this thesis. Firstly, transactional reconfiguration models apply a reconfiguration execution that does not take incoming events into consideration, regardless of their timeliness constraints. In these reconfiguration models, the processing deadlines of events are met only if the time to complete a reconfiguration falls within this deadline. Secondly, preemptive reconfiguration models apply a reconfiguration execution that directly interrupts an ongoing reconfiguration to process the incoming event. These models always meet an event's deadline, however, if there is a high event arrival rate, a reconfiguration completion might be delayed indefinitely. To adequately address these limitations, a new reconfiguration execution model is needed.

A time-adaptive reconfiguration model should support the dynamic adaptation of an ongoing reconfiguration process itself, as demanded by dynamic time bounds imposed by incoming events, while making progress towards reconfiguration completion. A primary challenge of this work is to maintain the dependency relationships between software entities in the presence of partially executed reconfigurations. Chapter 3 described the design of TimeAdapt, which follows a time-adaptive execution for its reconfigurations. TimeAdapt is designed for embedded software modelled according to the reconfigurable dataflow system model (RDF). In this model, entities, so-called actors, send data in a non-blocking manner and read data in a blocking manner. The definition of the reconfiguration model on this kind of software stems from two rationales: Firstly, the RDF system model has a strong theoretical background and its abstract definition allows the potential implementation of TimeAdapt for a variety of embedded platforms. Secondly, the RDF represents software that is deployed on a single processor platform. This choice is for scoping reasons, and to simplify the system model in this version of TimeAdapt. The focus of the work was on the complexity of the time-adaptive reconfiguration model. TimeAdapt itself leverages existing synchronisation mechanisms, such as bringing all affected software entities into a reconfiguration-safe state, and sequentially executing reconfiguration actions to guarantee a functioning system before and after the reconfiguration. The main contribution of the approach is the use of a deadline-aware scheduling mechanism that decides whether to execute the next reconfiguration action or to process an incoming event. The partial execution of a reconfiguration means that most event processing deadlines can be met, since the overall reconfiguration does not need to be completed. However, the model does not directly preempt an ongoing reconfiguration action, but completes at least the currently executing reconfiguration action. The incremental execution of reconfiguration actions leads to an eventual completion of a reconfiguration. However, because TimeAdapt favours an ongoing reconfiguration over a direct processing of an incoming event, it cannot guarantee the meeting of event deadlines, especially when deadlines are smaller than reconfiguration action execution durations. As a result, the model cannot be used for embedded software that has strict deadlines on its incoming events. TimeAdapt implements two scheduling algorithms that realise the deadline-aware scheduling mechanism and that differ in the granularity of the reconfiguration actions scheduled. The pessimistic scheduling algorithm considers safe steps as atomic units and schedules them only, if the overall execution duration is within the processing deadline. The optimistic scheduling algorithm considers individual reconfiguration actions, with the possibility of a revoke of this action, if the execution duration seems to exceed the processing deadline. TimeAdapt maintains dependency relationships between software entities by partitioning the remaining reconfiguration sequence into sub-sequences that need to be executed atomically, so-called safe steps. These safe steps are determined by the reconfiguration designer at reconfiguration design time and are input to the reconfiguration manager, which then executes the safe steps at reconfiguration execution duration.

The implementation of TimeAdapt was described in Chapter 4. The set of techniques and algorithms of the reconfiguration model were implemented in a manner that facilitates extensibility, such as the introduction of new reconfiguration action types. As the implementation is Java-based, it can be mapped to various embedded platforms.

The entities of the abstract dataflow system model are mapped to a Java-based component model, TimeAct. The TimeAct component model was implemented for this thesis because there are no component models targeting embedded software that realise the dataflow computational model. This component model realises a layered approach, in which interfaces realise different functional and non-functional concerns of a component, such as reconfiguration. TimeAdapt is implemented for TimeAct components. The central part of TimeAdapt's implementation is the reconfiguration manager, which has access to all components and executes reconfigurations on these components.

The evaluation of TimeAdapt on a real embedded platform, Java SunSpots, was described in Chapter 5. A sensing application was implemented, using the TimeAct component model and five experiments were conducted against the implementations of TimeAdapt's two scheduling algorithms, as well as implementations of a transactional and a preemptive reconfiguration model. The experiments differ in the values of parameters used, such as reconfiguration action execution durations, event processing deadlines, and event arrival rates. Values for these parameters are taken either directly from the underlying platform, or from existing work that evaluates software on the specific platform. The results highlight that TimeAdapt outperforms the implementation of a transactional reconfiguration model in terms of percentage of event processing deadlines met. This holds for an increase in the reconfiguration deadlines and varying event arrival rates. However, the experiment also illustrated that TimeAdapt cannot guarantee the meeting of all its deadlines and is only suitable for embedded software in which an occasionally missed event is tolerable. TimeAdapt also outperforms the implementation of a preemptive reconfiguration model with respect to the percentage of remaining reconfiguration actions, indicating that it makes more progress with an ongoing reconfiguration. The experiment showed that in settings where the event arrival rate is very high, and the associated event deadlines are smaller than the reconfiguration action execution durations, TimeAdapt encounters the same issue with reconfiguration starvation than a preemptive reconfiguration model. The results also illustrate that due to its fine-grained scheduling, TimeAdapt's optimistic algorithm has a higher percentage of remaining reconfiguration actions than the pessimistic mode.

As one would expect, the execution duration of a reconfiguration action has an impact on the percentage of deadlines met and the percentage of remaining reconfiguration actions. A reconfiguration sequence comprised of actions with a short execution duration meets a higher percentage of deadlines (1% to 50%), compared to sequences that are comprised of reconfiguration actions with a high execution duration. This is because actions need to be executed atomically, and TimeAdapt can return faster from a short reconfiguration action to process an event compared to when reconfiguration actions have high execution durations. Similarly, the execution duration of a safe step influences TimeAdapt's pessimistic mode and its meeting of event deadlines. The higher the execution duration of a safe step, the higher the deadline needs to be so that it can be met, as safe steps need to be executed atomically. The number of reconfiguration actions within a safe step has an influence on the percentage of remaining reconfiguration actions. The more actions are in a safe step and need to be executed atomically, the higher the given throughput of reconfiguration actions. An increasing number of event sources also impacts TimeAdapt's performance, as the higher the number of event sources, the higher the number of occurring events, resulting in a higher number of queued events that need to be processed. Queued events have an increased waiting time until they can be processed and hence a lower percentage of deadlines met and a lower percentage of remaining reconfiguration actions for the same given deadline than when a single event source is applied. The evaluation also investigated the reconfiguration overhead of TimeAdapt on the overall execution duration of a reconfiguration sequence. The worst case overhead is 30%, however, the most frequently observed overhead is in the range of 2%.

In summary, the research presented in this thesis focussed on providing a reconfiguration model that allows the timely reaction to incoming events during an ongoing reconfiguration, while at the same time progress towards reconfiguration completion is made. At all times, the dependency relationships of software entities are maintained. The main contributions of this thesis are summarised as:

- An overview of existing reconfiguration models, targeting different kinds of embedded software, such as control system software, embedded operating system software, and middleware software. The models are evaluated with a particular focus on their execution model, and whether they guarantee the completion of a reconfiguration, and the processing of incoming events within associated deadlines. Selected concepts from these reconfiguration models, such as the definition on top of an abstract system model, as well as the adoption of a centralised reconfiguration manager that has access to all system entities, were influential for TimeAdapt's design.

- A dynamic approach to reconfiguration execution based on a deadline-aware scheduling mechanism. This mechanism supports the interruption of an ongoing reconfiguration when an event occurs and tries to meet the event's associated processing. Two realisations of the deadline-aware scheduling mechanism, which differ in the granularity of what constitutes an atomic sequence, show that this mechanism meets a higher percentage of deadlines than reconfiguration models that follow a transactional execution model. Additionally, the greedy, deadline-aware scheduling of reconfiguration actions ensures that reconfigurations are eventually completed.

- The maintenance of dependency relationships between software entities in the presence of partial reconfigurations. This is ensured by partitioning the reconfiguration sequence into so-called safe steps. Safe steps denote sets of reconfiguration actions that, when executed atomically, lead from a functioning configuration to a new, functioning configuration. Safe steps are specified by the reconfiguration designer at reconfiguration design time.

- The implementation and evaluation of TimeAdapt for a software scenario on a real embedded system platform. This evaluation shows that the proposed mechanisms are beneficial for a class of embedded software that has soft time constraints associated with its events. The model is compared to implementations of reconfiguration models that follow a transactional and a preemptive execution model. The evaluation confirms that a higher percentage of deadlines are met by TimeAdapt than by a transactional reconfiguration model. At the same time, TimeAdapt has a lower percentage of remaining reconfiguration actions, compared to a preemptive reconfiguration model.

The main limitations are:

- TimeAdapt's preference of an ongoing reconfiguration action over an incoming event leads to the potential miss of the event's deadline, particularly if its associated deadline is smaller than the reconfiguration action execution durations. As a result, the reconfiguration model is not suitable for embedded software that has hard deadlines associated with its events.

- TimeAdapt cannot guarantee the progress towards reconfiguration completion if the event arrival rate is very high, and event deadlines are smaller than reconfiguration action execution durations. In this case, the likelihood is high that an event occurs while another event is still being processed by TimeAdapt. If the deadline is smaller than a reconfiguration action, no reconfiguration actions, or phases, can be scheduled by TimeAdapt and the reconfiguration is aborted to process the incoming event.

- For scoping reasons, TimeAdapt targets a simplified embedded software model, in which the software itself is deployed on a single processor embedded platform.

6.2 Future Work

This section outlines the key areas identified for future work. We distinguish the future work into whether it applies to the theoretical reconfiguration model or to the TimeAdapt implementation, and list it as follows: Integration with a time-predictive statistical model; Support for non-dataflowbased computational models; Improvement of timing guarantees; Extension to TimeAdapt design and implementation.

6.2.1 Integration with a time-predictive statistical model

The current model assumes that each reconfiguration action has associated estimation methods that return the estimated execution durations of either the action itself, or its state transfer and update connection phases. However, as these estimated execution durations are statically defined, they do not take current operating conditions into account, which might affect the actual execution duration. Approaches that combine statistical models and runtime measurements to predict execution durations offer an interesting alternative. These approaches combine off-line measurements obtained by the system's previous execution duration history with run-time generated timings based on statistical models (Brennan et al., 2009).

6.2.2 Support for non-dataflow based computational models

TimeAdapt is defined for software entities that follow a dataflow-based computational model. The definition of when system configurations are safe and the maintenance of dependency relationships between software entities are based on the loose coupling of these entities. Future developments of TimeAdapt should investigate changes to the reconfiguration model that would be required to support it working on software entities that follow different computational models. For example, the publish-subscribe computational model also dictates a loose and asynchronous coupling between its software entities, and is used to describe a range of embedded software. Investigations into porting TimeAdapt for this computational model need to include how dependency relationships between software entities are defined in this computational model and how they can be maintained.

6.2.3 Improvement of Timing Guarantees

The current reconfiguration model does not differentiate whether an incoming event affects a software entity that is currently being reconfigured, or a software entity that is not subject to reconfiguration. However, the percentage of deadlines met can be improved if the underlying deadline-aware scheduling mechanism is extended to support scoping. Events that occur at a software entity that is currently not the subject of reconfiguration, and whose processing does not affect any entities being reconfigured, can be directly processed by pausing the currently active reconfiguration action.

6.2.4 Extension to TimeAdapt Design

TimeAdapt's underlying RDF system model has two intrinsic characteristics that allow it to model both local, and distributed software entities. Firstly, the local and remote bindings between software entities are semantically identical, as in both cases the entities are connected via a data-store. Secondly, the RDF model allows for multiple bindings between different nodes. However, the current TimeAdapt reconfiguration model assumes that all software entities reside on an embedded system with a single processor. This requires a centralised reconfiguration manager that has global access to all entities. A potential area for future work is to extend TimeAdapt for execution on distributed software entities. This requires new mechanisms to handle TimeAdapt for multiple, distributed reconfiguration managers. As part of this work, the synchronisation of reconfiguration actions between different reconfiguration managers and the effect of partial reconfigurations on the synchronisation process needs to be investigated.

Additionally, future work could investigate the combination of the pessimistic and optimistic mode, and in which experimental settings this combination outperforms the basic modes. The combination of the two modes needs additional information on the current load, such as monitoring the current event arrival rate, to determine, whether to switch the mode.

6.2.5 Extension to TimeAdapt Implementation

In TimeAct, future work includes the extension of the component base class to support different computational models, such as the publish-subscribe computational model. In addition, the component model should be extended to support hierarchical components. The introduction of hierarchical components requires that a reconfiguration designer has knowledge about the underlying component topology, as a reconfiguration sequence needs to refer explicitly to sub-components.

In terms of the reconfiguration model, a C-based implementation of the model and the TimeAct system model would allow the evaluation of the model on more resource-constrained platforms.

6.3 Summary

This chapter summarised the motivation for the research undertaken and the most significant achievements of the work presented in this thesis. It outlined how this work contributed to the state of the art in reconfiguration models targeting embedded software by providing a time-adaptive execution model. The timely processing of events is realised by the provision of mechanisms that allow the partitioning of a reconfiguration sequence into sub-sequences. A progress towards reconfiguration completion is realised by algorithms that ensure the maintenance of dependency relationships between software entities. Experiments conducted on a real embedded platform show that these techniques lead to a higher percentage of met deadlines and a faster reconfiguration completion than existing applied execution models. The chapter concluded with suggestions for future work arising from the research undertaken in relation to this thesis.

Appendix A

TimeAct Component Model Implementation

```
public abstract class IComponent {
/* Fields */
protected String nameId;
protected Hashtable inports;
protected Hashtable outports;

/* Methods */
public abstract void fire();
public abstract void start();
public abstract void stop();
public void setOutport(String s, IChannel c);
public Hashtable getOutports();
}
```

Listing A.1: IComponent class

```
public interface IComponentFactory {
public IComponent create(String typeId);
public void delete(String nameId);
}
```

Listing A.2: IComponentFactory interface

```
public interface Introspection {
public Object getInterface(String name);
public Object getPort(String name);
}
```

Listing A.3: Introspection interface

```
public interface IReconfiguration {
/* Behavioural Reconfiguration Methods */
public void transferState(IComponent actor, IComponent newActor);
public void upgradeConnections(IComponent actor, IComponent newActor);
public void execute(Action a);
public void make_quiescent();
/* Time Estimation Methods */
public double getEstimatedTimeUpdateConnections();
public double getEstimatedRevokeTime();
public double getExecutionTime(Action a);
}
```

Listing A.4: Reconfiguration interface

```
public interface IStateAccess {
public void set(String name, Object o);
public Object get(String name);
}
```

Listing A.5: StateAccess interface

```
public interface IChannel {
public void push(Object o);
public Object read();
}
```

Listing A.6: IChannel interface

Appendix B

TimeAdapt Reconfiguration Model

```
public class ReconfigurationManager implements ISystemConfiguration,
IArchitecturalReflectiveExtension,
IReconfigurationInitalisation, IReconfigurationExecution,
IEventProcessing, IComponentFactory {
/* Fields */
private Vector<IComponent> componentList;
private ReconfigurationAlgorithm algo;
private Graph graph;
private Vector<Interrupt> interruptList;
private ReconfigurationManager instance;
private EventQueue queue;

/* Methods: */
public static ReconfigurationManager getInstance() {...}
/*Realisation of ISystemConfiguration*/
/*Realisation of IArchitecturalReflectiveExtension*/
/*Realisation of IReconfigurationInitalisation*/
/*Realisation of IReconfigurationExecution*/
/*Realisation of IEventProcessing*/
/*Realisation of IComponentFactory*/
}
```

Listing B.1: ReconfigurationManager class

```
public interface ISystemConfiguration {
public String registerActor(Icomponent c);
public void connect(Icomponent a, IComponent b);
public void disconnect(IComponent a, IComponent b);
}
```

Listing B.2: System configuration interface

```
public interface IArchitecturalReflectiveExtension {
public Vector getNodes();
public IComponent getNode(String s);
}
```

Listing B.3: Architectural reflective extension interface

```
public interface IReconfigurationInitalisation {
public Graph getReconfigurationGraph();
public Graph generateSequence(Vector<ReconfigurationAction> s);
}
```

Listing B.4: Reconfiguration initalisation interface

```
public interface IReconfigurationExecution {
public void executeReconfiguration();
public void updateAllConnections(IComponent oldC, IComponent newC);
}
```

Listing B.5: Reconfiguration execution interface

```
public interface IEventProcessing {
public void notifyInterrupt(double deadline);
}
```

Listing B.6: Event processing interface

```
public abstract class ReconfigurationAlgorithm {
/*Fields */
ReconfigurationManager rm;

/* Methods */
public abstract Graph executeReconfiguration(Graph g, double deadline);
}
```

Listing B.7: ReconfigurationAlgorithm class

```
public class Interrupt {
/*Fields */
private String type;
private double responseDeadline;

/* Methods */
public Interrupt(String t, double d) {...}
}
```

Listing B.8: Interrupt class

```
public abstract class ReconfigurationAction {
/*Fields */
IComponent actor;
/* Methods */
public IComponent getActor() { return actor ;}
public abstract IComponent getNextActor();
public abstract Action clone();
public abstract double getEstimatedTimeUpdateConnections();
public abstract double getEstimatedRevokeTime();
public abstract double getExecutionTime();
public abstract void execute();
}
```

Listing B.9: Reconfiguration action class

```
public class Graph {
/*Fields */
Vector<IComponent>[] safeSteps;
/* Methods */
public Vector<ReconfiguratioonActions>
partition(Vector<ReconfigurationActions> a);
public Edge addEdge(SafeStep p1, SafeStep p2);
public Vector getNext();
public void addAction(Action a);
}
```

Listing B.10: Reconfiguration action graph class

Appendix C

Detailed Evaluation Results

C.1 Reconfiguration Execution Times

The following tables summarise the reconfiguration execution times that were used in experiment 1 to calculate the percentage of deadlines met. All tables are of the form:

Deadlines [ms]
Pessimistic TA Mean Execution Time [ms]
Optimistic TA Mean Execution Time [ms]
Transactional RM Mean Execution Time [ms]
Pairwise T-Value (Pessimistic vs. Transactional)
Pairwise T-Value (Optimistic vs. Transactional)

0.15	10	20	30	40	50	60	70	80	90	100
12.42	12.31	12.38	16.79	25.26	41.11	48.95	47.81	54.69	61.40	62.73
8.36	8.44	8.88	18.78	26.05	40.77	48.22	50.02	52.70	58.38	56.89
69.35	69.28	72.55	72.35	68.72	74.76	73.44	72.30	74.16	75.42	70.09
-20.13	-20.71	-22.50	-19.42	-16.10	-14.47	-12.32	-12.02	-11.12	-9.97	-7.84
-21.65	-22.11	-23.90	-19.39	-16.47	-16.86	-15.10	-15.50	-15.81	-17.79	-22.85

Table C.1: Statistical values for low event arrival time

0.15	10	20	30	40	50	60	70	80	90	100
12.35	12.61	12.61	16.50	24.26	30.93	37.48	43.44	49.66	55.32	60.41
7.47	8.47	10.07	19.71	28.13	39.14	46.87	57.51	65.09	76.26	83.43
94.08	93.23	95.58	93.05	94.41	95.28	94.53	95.25	94.53	95.24	94.17
-74.13	-66.20	-80.24	-64.73	-66.96	-58.01	-56.41	-56.65	-49.79	-50.33	-55.70
-78.75	-69.99	-83.18	-65.44	-67.05	-51.84	-55.60	-45.89	-35.00	-13.06	-5.68

Table C.2: Statistical values for medium event arrival time

0.15	10	20	30	40	50	60	70	80	90	100
12.358	12.528	12.465	18.441	28.628	34.936	46.644	53.528	60.136	66.316	70.168
2.960	6.883	17.593	24.835	31.349	43.607	53.202	68.427	84.367	105.724	128.88
110.033	110.201	109.987	110.655	110.357	110.161	109.715	110.685	110.209	110.660	110.676
-369.7	-380.2	-398.3	-295.6	-271.4	-304	-154.9	-193.7	-152	-106.6	-117.9
-394.3	-383.1	-344.5	-354.6	-254.2	-178.9	-109.2	-68.3	-40.9	-6.2	-16.4

Table C.3: Statistical values for high event arrival time

C.2 Percentage of Reconfiguration Actions Remaining

The following tables summarise the detailed results for reconfiguration actions remaining obtained in experiment 1. All tables in this chapter are of the form:

Deadlines [ms]
Pessimistic TA Mean Percentage of Actions Remaining [%]
Optimistic TA Mean Percentage of Actions Remaining [%]
Preemptive RM Mean Percentage of Actions Remaining [%]
Pairwise T-Value (Pessimistic vs. Transactional)
Pairwise T-Value (Optimistic vs. Transactional)

0.15	10	20	30	40	50	60	70	80	90	100
41.48	45.46	42.38	44	33.24	28.22	20.5	16.74	13.36	6.54	5.06
60.06	57.22	52.10	45.92	41.58	37.96	30.6	27.22	15.32	8.9	4.62
63.56	64.06	65.36	60.66	59.76	60.6	63.12	65.92	64.86	65.16	60.66
-5.81	-5.5	-7.27	-4.19	-6.54	-11.19	-13.38	-17.95	-16.88	-21.16	-21.36
-0.95	-2.03	-3.58	-4.75	-5.46	-8.13	-11.99	-14.72	-19.58	-22.80	-21.96

Table C.4: Statistical values for low event arrival rate

0.15	10	20	30	40	50	60	70	80	90	100
40.58	41.42	43.0	38.8	31.78	25.1	21.92	16.92	14.13	10.27	6.06
57.79	56.59	49.58	46.35	40.76	34.20	29.9	24.49	17.78	15.15	6.2
60.34	61.85	62.59	60.14	62.34	59.36	60.52	60.84	61.52	61.1	61.54
-13.54	-15.43	-13.87	-16.06	-22.91	-26.86	-30.82	-41.35	-41.41	-49.97	-54.65
-1.87	-4.13	-9.79	-9.19	-14.2	-18.34	-23.20	-29.59	-34.30	-51.66	-50.02

Table C.5: Statistical values for medium event arrival rate

0.15	10	20	30	40	50	60	70	80	90	100
41.06	40.33	40.70	36.97	31.81	26.49	21.60	18.67	12.55	6.75	3.09
59.9	56.97	50.71	50.06	43.55	39.0	30.42	27.37	25.17	19.94	14.02
60.2	60.4	60.0	60.0	59.85	60.24	60.19	61.23	60.07	59.9	59.9
-45.76	-40.79	-41.56	-45.37	-61.14	-91.94	-97.98	-118.4	-102.5	-128.8	-156.4
-0.84	-8.24	-19.52	-22.04	-27.51	-47.76	-52.99	-64.17	-68.49	-83.71	-94.1

Table C.6: Statistical values for high event arrival rate

Bibliography

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press.

Aksit, M., & Choukair, Z. (2003). Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*. Providence, Rhode Island, USA.

Almeida, a. P. A., Jo Van Sinderen, M., & Nieuwenhuis, L. (2001). Transparent dynamic reconfiguration for CORBA. In *DOA '01: Proceedings of the Third International Symposium on Distributed Objects and Applications*. Rome, Italy.

Baresi, L., Di Nitto, E., & Ghezzi, C. (2006). Toward open-world software: Issue and challenges. *Computer*, *39*(10), 36–43.

Brennan, S., Cahill, V., & Clarke, S. (2009). Applying non-constant volatility analysis methods to software timeliness. In *12th Euromicro Conference on Real-Time Systems, Work-in-progress Session*. Dublin, Ireland.

Brinkschulte, U., Krakowski, C., Riemschneider, J., Kreuzinger, J., Pfeffer, M., & Ungerer, T. (2000). A microkernel architecture for a highly scalable real-time middleware. In *RTAS 2000, 6th IEEE Real-time Technology and Application Symposium, Work in Progress session*. Washington, DC, USA.

Cazzola, W., Savigni, A., Sosio, A., & Tisato, F. (1998). Architectural reflection: Bridging the gap between a running system and its architectural specification. In *In proceedings of 6th Reengineering Forum (REF'98*, (pp. 8–11). IEEE.

Cheong, E. (2003). Design and implementation of tinyGALS: A programming model for event-driven embedded systems. Tech. rep., Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, USA.

Cheong, E. (2007). *Actor-Oriented Programming for Wireless Sensor Networks*. Ph.D. thesis, Electical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, USA.

Cheong, E., Liebman, J., Liu, J., & Zhao, F. (2003). TinyGALS: A programming model for event-driven embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*. Melbourne, Florida, USA.

Cheong, E., & Liu, J. (2005). galsC: A language for event-driven embedded systems. In *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*. Munich, Germany.

Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G. P., Sivaharan, T., Weerasinghe, N., & Zachariadis, S. (2007). The runes middleware for networked embedded systems and its application in a disaster management scenario. In *PERCOM '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*. White Plains, NY, USA.

Coulouris, G. F., Dollimore, J., & Kindberg, T. (2005). *Distributed systems: concepts and design*. Pearson Education.

Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., & Sivaharan, T. (2008). A generic component model for building systems software. *ACM Transactions on Computing Systems*, *26*, 1–42.

Crossbow Technology Inc. (2004). Mica Motes. Website.
URL `http://www.xbow.com/`

David, P.-C., & Ledoux, T. (2006a). An aspect-oriented approach for developing self-adaptive fractal components. In *International Conference on Software Composition*. Vienna, Austria.

David, P.-C., & Ledoux, T. (2006b). Safe dynamic reconfigurations of Fractal architectures with FScript. In *Proceedings of the CBSE Workshop, ECOOP*. Nantes, France.

Dowling, J. (2004). *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. Ph.D. thesis, Department of Computer Science, Trinity College Dublin, Dublin, Ireland.

Dunkels, A., Grnvall, B., & Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*. Tampa, Florida, USA.

Friedrich, L. F., Stankovic, J., Humphrey, M., Marley, M., & Haskins, J. (2001). A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro*, *21*(3), 54–68.

Ghiasi, S., Nahapetian, A., Moon, H. J., & Sarrafzadeh, M. (2005). Reconfiguration in network of embedded systems: Challenges and adaptive tracking case study. *Journal of Embedded Computing*, *Volume 1, Number 1/2005*, 147–166.

Goldman, R. (2009). Using the at91 timer/counter. Website.
URL `http://www.sunspotworld.com/docs/AppNotes/TimerCounterAppNote.pdf`

Goudarzi, K. M. (1999). *Consistency preserving dynamic reconfiguration of distributed systems*. Ph.D. thesis, Imperial College, London, UK.

Goudarzi, K. M., & Kramer, J. (1996). Maintaining node consistency in the face of dynamic change. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*. Annapolis, Maryland, USA.

Grace, P. (2008). *Dynamic Adaptation in Middleware for Network Eccentric and Mobile Applications*, chap. 13, (pp. 285–302). Springer.

Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers.

Hammer, M. (2009). *How to Touch a Running System - Reconfiguration of Stateful Components*. Ph.D. thesis, Ludwig-Maximilian-Universitaet, Munich, Germany.

Han, C.-C., Kumar, R., Shea, R., Kohler, E., & Srivastava, M. (2005). A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. Seattle, WA, USA.

Hillman, J., & Warren, I. (2004). Quantitative analysis of dynamic reconfiguration algorithms. In *International Conference on Design, Analysis, and Simulation of Distributed Systems (DASD)*. Virginia, USA.

Hofmeister, C. (1994). *Dynamic Reconfiguration of Distributed Applications*. Ph.D. thesis, University of Maryland, College Park, USA.

Huang, K., Santinelli, L., Chen, J.-J., Thiele, L., & Buttazzo, G. C. (2009). Adaptive dynamic power management for hard real-time systems. In *RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*. Washington, D.C., USA.

Hughes, D., Thoelen, K., Horré, W., Matthys, N., del Cid Garcia, P. J., Michiels, S., Huygens, C., & Joosen, W. (2009). LooCi: A loosely-coupled component infrastructure for networked embedded systems. In *Proceedings of the 7th International Conference on Advances in Mobile Computing & Multimedia,*. Colmar, France: ACM.

Hughes, D., Thoelen, K., Horré, W., Matthys, N., Michiels, S., Huygens, C., Joosen, W., & Ueyama, J. (2010). Building wireless sensor network applications with LooCi. *The International Journal of Mobile Computing and Multimedia Communications (IJMCMC)*.

Issel, H. (2006). *Dynamische Rekonfiguration in eingebetteten Regelungssystemen*. Master's thesis, Hasso-Plattner-Institut fuer Softwaresystemtechnik, Universitaet Potsdam, Germany.

Janssens, N. (2006). *Dynamic Software Reconfiguration in Programmable Networks*. Ph.D. thesis, Katholieke Universiteit Leuven, Leuven, Belgium.

Janssens, N., Joosen, W., & Verbaeten, P. (2005). Necoman: Middleware for safe distributed-service adaptation in programmable networks. *IEEE Distributed Systems Online*, *6*(7).

Kon, F., Costa, F., Blair, G., & Campbell, R. H. (2002). The case for reflective middleware. *Communications of the ACM*, *45*(6), 33–38.

Kopetz, H. (1997). *Real-Time Systems Design Principles for Distributed Embedded Systems*. Kluwer Academic Publishers.

Kramer, J., & Magee, J. (1985). Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, *11*(4), 424–436.

Kramer, J., & Magee, J. (1990). The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, *16*(11), 1293–1306.

Lau, K.-K., & Wang, Z. (2005). A taxonomy of software component models. In *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*. Porto, Portugal.

Lee, E. A. (2000). What's ahead for embedded software? *Computer*, *9*(33), 18–26.

Lee, E. A. (2002). Embedded software. In *Advances in Computers, Vol 56*, (p. 2002). Academic Press.

Lee, E. A., Neuendorffer, S., & Wirthlin, M. J. (2003). Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, *12*, 231–260.

Léger, M., Ledoux, T., & Coupaye, T. (2007). Reliable dynamic reconfigurations in the fractal component model. In *ARM '07: Proceedings of the 6th international workshop on Adaptive and reflective middleware*. Newport Beach, California, USA.

Li, W. (2009). Dynaqos-rdf: a best effort for qos-assurance of dynamic reconfiguration of dataflow systems. *Journal of Software Maintenance and Evolution*, *21*(1), 19–48.

Liu, J. W. S. (2000). *Real-Time System*. Prentice Hall.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., & Cheng, B. H. C. (2004). Composing adaptive software. *Computer*, *37*(7), 56–64.

Michiels, S. (2003). *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. Ph.D. thesis, Katholieke Universiteit Leuven, Leuven, Belgium.

Mitchell, S., Naguib, H., Coulouris, G., & Kindberg, T. (1998). Dynamically reconfiguring multimedia components: A model-based approach. In *Proceedings of the 8th ACM SIGOPS European Workshop, Sintra, Portugal*. Sintra, Portugal.

Mitchell, S., Naguib, H., Coulouris, G., & Kindberg, T. (1999). A QoS support framework for dynamically reconfigurable multimedia applications. In *Proceedings of the IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems II*, (pp. 17–30). Deventer, The Netherlands.

Neamtiu, I., & Hicks, M. (2009). Safe and timely updates to multi-threaded programs. *ACM SIGPLAN Notices*, *44*(6), 13–24.

Object Management Group (1999). OMG IDL specificiation. Website.
URL `http://www.omg.org/gettingstarted`

OSGI Alliance (2009). OSGI. Website.
URL `http://www.osgi.org/Main/HomePage`

OW2 Consortium (1999). Fractal Component Model. Website.
URL `http://fractal.ow2.org/`

Perrson, M. (2009). *Adaptive Middleware for Self-Configurable Embedded Real-time Software*. Ph.D. thesis, KTH Stockholm, Stockholm, Sweden.

Polakovic, J., Ozcan, A., & Stefani, J.-B. (2006). Building reconfigurable component-based os with think. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, (pp. 178–185). Cavtat/Dubrovnik, Croatia.

Polakovic, J., & Stefani, J.-B. (2008). Architecting reconfigurable component-based operating systems. *Journal of System Architecture*, *54*(6), 562–575.

Popovici, A., Gross, T., & Alonso, G. (2002). Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. Enschede, The Netherlands.

Rasche, A., & Polze, A. (2003). Configuration and dynamic reconfiguration of component-based applications with microsoft .net. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Hakodate, Hokkaido, Japan.

Rasche, A., & Polze, A. (2005). Dynamic reconfiguration of component-based real-time software. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. Sedona, Arizona, USA.

Rasche, A., & Polze, A. (2008). Redac dynamic reconfiguration of distributed component-based applications with cyclic dependencies. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, (pp. 322–330). Orlando, Florida, USA.

Real, J., & Crespo, A. (2004). Mode change protocols for real-time systems: A survey and new proposal. *Real-time Systems*, *26*, 161–197.

Regehr, J. (2008). *Safe and Structured Use of Interrupts in Real-Time and Embedded Software*, chap. 16. Chapman and Hall.

Rutten, E. (2008). Reactive control of adaptive embedded systems: a position paper. In *ARM '08: Proceedings of the 7th workshop on Reflective and adaptive middleware*. Leuven, Belgium.

Schmidt, D. C. (2002). Middleware for real-time and embedded systems. *Communication of the ACM*, *45*(6), 43–48.

Schneider, E. (2004). *A Middleware Approach For Real-Time Software Reconfiguration on Distributed Embedded Systems*. Ph.D. thesis, Universite Louis Pasteur Strasbourg, Strasbourg, France.

Schneider, E., Picioroagă, F., & Brinkschulte, U. (2004). Dynamic reconfiguration through osa+, a real-time middleware. In *DSM '04: Proceedings of the 1st international doctoral symposium on Middleware*. Toronto, Canada.

Seto, D., Krogh, B., Sha, L., & Chutinan, A. (1998). The simplex architecture for safe online control system upgrades. In *Proceedings of the American Control Conference*, vol. 6, (pp. 3504–3508). Philadelphia, PA, USA.

Shaw, M., & Garlan, D. (1996). *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

Simon, D., Cifuentes, C., Cleal, D., Daniels, J., & White, D. (2006). Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*. Ottawa, Canada.

Soules, C. A. N., Appavoo, J., Hui, K., Wisniewski, R. W., Silva, D. D., Ganger, G. R., Krieger, O., Stumm, M., Auslander, M., Ostrowski, M., Rosenburg, B., & Xenidis, J. (2003). System support for online reconfiguration. In *Proceedings of the Usenix Technical Conference*. San Antonio, TX, USA.

Stewart, D., & Khosla, P. (1996). The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering*, *6*, 249–277.

Stewart, D., Volpe, R., & Khosla, P. (1997). Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, *23*(12), 759–776.

Stewart, D. B., & Arora, G. (1996). Dynamically reconfigurable embedded software - does it make sense? In *ICECCS '96: Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems*. Montreal, Canada.

Stewart, D. B., Schmitz, D. E., & Khosla, P. K. (1992). The chimera ii real-time operating system for advanced sensor-based robotic applications. *IEEE Transactions on Systems, Man, and Cybernetics*, (pp. 1282–1295).

Strunk, E. A., & Knight, J. C. (2004). Assured reconfiguration of embedded real-time software. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*. Florence, Italy.

Sun (2006). Sun small programmable object technology (sun SPOT) owner's manual.

Website.
URL `http://www.sunspotworld.com/docs/Green/SunSPOT-OwnersManual.pdf`

Sun (2009). Java ME API. Website.
URL `http://java.sun.com/javame/reference/apis.jsp`

Tešanović, A., Amirijoo, M., Nilsson, D., Norin, H., & Hansson, J. (2005). Ensuring real-time performance guarantees in dynamically reconfigurable embedded systems. In *EUC 2005: Proceedings of the International Conference for Embedded and Ubiquitous Computing*, (pp. 131–141). Nagasaki, Japan.

Trapp, M., Adler, R., Förster, M., & Junger, J. (2007). Runtime adaptation in safety-critical automotive systems. In *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*. Innsbruck, Austria.

Truyen, E., Janssens, N., Sanen, F., & Joosen, W. (2008). Support for distributed adaptations in aspect-oriented middleware. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*. Brussels, Belgium.

Vandewoude, Y. (2007). *Dynamically updating component-oriented systems*. Ph.D. thesis, Katholieke Universiteit Leuven, Leuven, Belgium.

Wegdam, M. (2003). *Dynamic Reconfiguration and Load Distribution in Component Middleware*. Ph.D. thesis, University of Twente, Twente, The Netherlands.

Welch, I., & Stroud, R. J. (2000). Kava - a reflective java based on bytecode rewriting. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, (pp. 155–167). London, UK.

Wermelinger, M. (1997). A hierarchic architecture model for dynamic reconfiguration. In *PDSE '97: Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*. Boston, MA, ,USA.

Yi, S., Min, H., Cho, Y., & Hong, J. (2008). Molecule: An adaptive dynamic reconfiguration scheme for sensor operating systems. *Computer Communications*, *31*(4), 699–707.

Zhang, J., Chen, B., Yang, Z., & McKinley, P. (2005). Enabling safe dynamic component-based software adaptation. In *Architecting Dependable Systems Vol 3, Springer Lecture Notes for Computer Science*, (pp. 194–211).

Zhao, Z., & Li, W. (2007a). Dynamic reconfiguration of distributed data flow systems. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*. Bejing, China.

Zhao, Z., & Li, W. (2007b). Influence control for dynamic reconfiguration. In *ASWEC'07: Proceedings of the 2007 Australian Software Engineering Conference*. Melbourne, Australia.