Towards an Algebra for Unifying Theories of Concurrent Programming (UTCP)

Andrew Butterfield[1 [0000−0002−2337−2101]]

Trinity College Dublin, Ireland
butrfeld@tcd.ie

Abstract. Unifying Theories of Concurrent Programming (UTCP) is a denotational semantics of shared-variable concurrency, expressed using the notation and methodology of Unifying Theories of Programming (UTP). A key feature is that it is compositional, in that the semantics of a composite is described in terms of the semantics of its sub-components. The underlying language used is that which is used to define a Concurrent Kleene Algebra (CKA). This includes the notions of skip, atomic actions, iteration, and non-deterministic, sequential, and parallel composition. This chapter makes progress toward proving that UTCP satisfies the CKA laws. We describe the methodology used, and give proofs for key ideas, as well as proofs and proof-sketches for many of the laws. We also discuss open issues, the most notable being the precise nature of *miracle* in this setting. The chapter finishes with a roadmap of how we might encode well-established UTP concepts such as Designs and Reactive Systems on top of UTCP.

Keywords: Unifying Theories of Concurrent Programming · Concurrent Kleene Algebra · Denotational Semantics

1 Introduction

An Appreciation

My first academic encounter with Jim Woodcock was at FME'93 in Odense, when I presented my first Formal Methods paper [4]. I got actively involved with Formal Methods Europe at that point, and attended many of their meetings in subsequent years. I have happy memories of sitting in airport waiting lounges with him on our returns journeys, discussing the merits, etc, of various formalisms. This lead to a decision, at FM99 in Toulosue, that we would start an active collaboration. We started with the semantics of Handel-C [11,12] and this led to many publications in this space. A key turning point was when I spent 6 months on sabbatical with him in 2003, in Kent. Here I was introduced to what has now become my research paradigm of choice: Unifying Theories of Programming (UTP) [17]. This led to work looking at ways to add time-slots [9] to the *Circus* language he worked on with Ana Cavalcanti [28]. Other joint work included: Flash Memory [7] ; POSIX Grand Challenge Roadmap [16] ; and

Heterogenous Semantics [29]. He has always been a good friend and great inspiration for me over the years. He even played a key role in the more recent direction of my research by pointing me to the Views paper by Dinsdale-Young et. al. [14], which inspired me to go on my current deep dive into shared-variable concurrency. Thank you Jim!

1.1 Introduction to UTCP

Unifying Theories of Concurrent Programming (UTCP) is a denotational semantics of shared-variable concurrency, expressed using the notation and methodology of Unifying Theories of Programming (UTP). This theory was developed out of prior research into lightweight languages for loosely describing business process flows that included non-deterministic choice as a feature [8,25]. A key point was the Views paper by Dinsdale-Young et. al. [14], that introduced the notion of a View as a particular Semigroup and/or Monoid, abstracting state and composition, that characterised generic concurrent behaviour. They demonstrated how a wide range of concurrency reasoning approaches could all be instantiated as Views. These approaches include: Owicki-Gries, Rely-Guarantee, Separation Logic, and various Type systems, among others. The most abstract/general view is based on the same shared-state concurrency language that was used by Hoare and colleagues when they defined Concurrent Kleene Algebra (CKA) [20]. This also coincided with the loosest version of the business process flows language we had been exploring.

The motivation for this work has been to develop a compositional (denotational) semantics for this base language, in UTP, using a homogeneous alphabet where s is the before-state, and s' is the after-state. It turned out that some static observables needed to be added to manage necessary context information in order for this to work [6]. The aim of this chapter is to demonstrate how we might prove that the UTCP semantics satisfies the CKA laws.

1.2 Structure of this chapter

In Section 2 we summarise the language and semantics of UTCP, and present the laws of CKA, using the UTCP notation. Then, in Section 3 we describe our overall proof methodology, describing the key concepts needed to construct the correctness arguments. We then showcase proofs of key laws in Section 4, going to details of the reasonings. Finally, in Section 5 we discuss related work, and then we conclude in Section 6.

2 Background

2.1 UTP

The Unifying Theories of Programming framework [18] uses predicate calculus to define before-after relationships over appropriate collections of free observation

variables. The before-variables are undashed, while after-variables have dashes. A simple approach would be to simply observe the values of program variables, in which case the before- and after-*values* of program variable v would be represented by observational variables v and v' respectively. For example, the meaning of an assignment statement might be given as follows:

$$x := e \quad \widehat{=} \quad x' = e \wedge \nu' = \nu$$

The definition says that, *if* the assignment terminates, then the final value of variable x will be set equal to the value of expression e in the before-state, while the other variables, denoted collectively here by ν, remain unchanged. This leads to a theory of partial correctness for imperative programs.

The theory can be extended to cover total correctness by introducing Boolean observations of program starting (ok) and termination (ok'). In this case, we find that we need a technique that allows us to identify predicates whose interpretation is nonsense, and eliminate them from any semantic theory we might construct. For example, the predicate $\neg ok \wedge ok'$ describes a situation in which a program has not started, but has terminated.

In UTP we use the concept of healthiness conditions to specify which predicates are meaningful in the context of our theory. For the total correctness theory to work, we need to ensure that all predicates have the form $ok \wedge P \implies ok' \wedge Q$, where P and Q do not refer to ok or ok'. This is interpreted as saying, if the program is started and P holds true at the start, then the program will terminate with Q being satisfied at the end.

A standard UTP approach is to define healthy predicates as being fixed-points of suitable idempotent, monotonic predicate transformers. For example, in the total correctness theory, we can define a predicate transformer $\mathbf{H}(P) \widehat{=} ok \implies P$. A predicate D that satisfies $D = (ok \implies D)$ is one that only asserts its behaviour once it is started ($ok = \mathbf{true}$). Our healthiness conditions (Sec. 2.2) are expressed in this fashion.

An important characteristic of both the UTP theories referred to above, is that their predicates are interpreted as a relation between the before-state and after-state of a *complete* program execution. The "standard" treatment of process-algebra concurrency (e.g. ACP,CCS,CSp) in UTP [18, Chps. 7,8], is focussed on a model where parallel processes work with their own copy of global variables (if any), and where a merge function is used to merge their values once the parallel construct terminates. The observables include ok, $wait$, tr, and ref.

2.2 UTCP

The Unifying Theory of Concurrent Programming (UTCP) that we developed [6] formalises shared-variable concurrency, also as a relation between before- and after-states. It was derived by adapting Woodcock and Hughes Unifying Theories of Parallel Programming (UTPP) [30], in which statement syntax had labels, and the program state consisted of variable state s, as well as a set ls of labels of currently active statements, and semantics was defined as a relation

between ls, s and ls', s'. A statement would only be ready to run when its label was in ls, and once it had run, it would replace its own label (in ls') with the label of the following statement. A key part of our adaptation was adding a label generator g, designed to ensure that all statement labels were unique by construction. We also provided observation variables *in* and *out* to denote the starting and ending labels of constructs. The three new variables, g, *in*, and *out*, also differ from s and ls in that they did not have dashed counterparts. They are *static* observables. Their role is to manage the relationships between composite commands, and their sub-components. This means that we have a non-homogeneous alphabet $(g, in, out, s, ls, s', ls')$. A key paper in this regard is one by Lamport [24] which we did not encounter until this work was largely done. In it he developed an axiomatic semantics for concurrency, and describes some key principles that must hold for such a semantics to work. A key one is being able to talk both about one component in its own right, but also about any use of it as a subcomponent. This aspect is handled by our use of g, in, out.

Here we briefly summarise key UTCP concepts such as labels and their generators, observation variables, atomic actions, healthiness conditions, and semantic definitions. A fuller motivation and discussion can be found in [6].

Labels In order to manage flow-of-control, we need to be able to identify when every construct starts, is running, and ends. We adopted the idea from [30] that flow of control is managed by an auxiliary variable ls whose value is the set of all labels of constructs that are able to execute. Given some notion of labels ($l \in Lbs$), we introduce a generator ($g \in Gen$) that supports two operations: $new : Gen \to Lbl \times Gen$ that produces a new label and a new generator; while $split : Gen \to Gen \times Gen$ splits a generator into two new ones. In all cases we require that any labels obtained from new generators will not have been obtained previously from any of their parent generators.

To avoid long nested calls of *new*, *split* and projections π_1, π_2, we defined the following terse label and generator expression syntax:

$$\begin{aligned} g \in GVar &\qquad \text{The } \textit{sole} \text{ Generator variable} \\ G \in GExp &::= g \mid G_: \mid G_1 \mid G_2 \\ L \in LExp &::= \ell_G \end{aligned}$$

Here $G_:$ denotes the generator left once *new* has been run on G, with ℓ_G denoting the label so generated: $(\ell_G, G_:) = new(G)$. Expressions G_1 and G_2 denote the two outcomes of applying *split* to G: $(G_1, G_2) = split(G)$. Effectively we have $\__:$, $\__1$, and $\__2$ as postfix operators acting on generators. We use $labs(G)$ to denote all the labels that G can generate and we require the following laws to hold:

$$\begin{aligned} labs(G) &= \{\ell_G\} \cup labs(G_:) \cup labs(G_1) \cup labs(G_2) \\ \ell_G &\notin labs(G_:) \cup labs(G_1) \cup labs(G_2) \\ \emptyset &= labs(G_1) \cap labs(G_2) \end{aligned}$$

This language allows us to take a generated label expression like

$$\pi_1(new(\pi_2(new(\pi_2(split(\pi_2(new(\pi_1(split(g)))))))))).$$

and replace it with $\ell_{g1:2:}$. This notation is compact, and may appear very contrived. However it has one very strong advantage: it makes generators and their labels "relocatable", in much the same way as some program code can be so considered. The variable g can be viewed as a sort of "base", with all of the labels generated from it being relative to that base. We can do this, in one way only, by substituting any generator expression for g. If we replace g with something different, then we "shift" all the associated labels accordingly. If γ and σ range over sequences of :, 1 and 2, then

$$(\ell_{g\gamma})[g\sigma/g] = \ell_{g\sigma\gamma} \tag{1}$$

In effect the substitution "relocates" generator g by running *new* and *split* on it as specified by σ, and any labels are in effect generated by this relocated generator using their γ specification. This simple use of substitution gives us a really easy way to compose program fragments in terms of their semantics. This relocation feature is a key aspect identified by Lamport [24].

Observations The values associated with all (shared) variables are not mentioned individually, but instead are lumped together; and we assume that all actions are labelled and that we can observe the set of labels that are considered to be "active".

$$s, s' : State \tag{2}$$
$$ls, ls' : \mathcal{P}\, Lbl \tag{3}$$

The role of label-generators is rather different, however. They will be used to generate labels for statements, and we do not want these to change during the lifetime of the program. We will also want to be able to refer in a general way to two key labels associated with any language construct, namely the label (*in*) that is used to enable the starting of a construct, and the label (*out*) that is used to signal that the construct has just terminated.

$$in, out : Lbl \tag{4}$$
$$g : Gen \tag{5}$$

These observations are *static*, in that their values do not change during program execution. Instead, these variables record context-sensitive information about how a language construct is situated with respect to its "neighbours", in a way that permits a compositional approach.

This brings us to an important distinction between the usual approach taken by UTP where a program syntax is simply a shorthand notation for its semantics. This "punning" between syntax and semantics largely works for theories

of sequential programs or local-state concurrency, mainly because sequences of code lead to simple semantic sequencing. However, in global shared-variable concurrency, code sequences get broken up by interference from parallel execution threads, and there is no longer a simple correspondence between syntactical and semantic sequencing. Here we shall use the notation $P;Q$ to denote *semantic* sequential composition, which means that the execution of P is immediately followed by the execution of Q, without any intervening external interference. It has precisely the standard UTP definition:

$$P;Q \mathrel{\widehat{=}} \exists s_m, ls_m \bullet P[s_m, ls_m/s', ls'] \wedge Q[s_m, ls_m/s, ls] \tag{6}$$

The key thing to note is that this definition makes no reference at all to g, in or out, as these are static observations. We also define state skip (ii) and *semantic* skip ($I\!I$), the unit for semantic sequential composition, as

$$\begin{aligned} ii &\mathrel{\widehat{=}} s' = s \\ I\!I &\mathrel{\widehat{=}} ls' = ls \wedge ii \end{aligned}$$

Another important consequence of having to deal with interference, is how we interpret the temporality aspect of s and ls. What precisely differs between an un-dashed variable (s,ls) and the dashed ones (s',ls')? In the basic model of sequential programming, as well as its reformulation using Designs, we interpret s as the value of the state *before* the program runs, and s' as the value *after* the program terminates.

Our move to shared-state concurrency takes a more nuanced view of s'. This is now still interpreted as being the value of the state after the program finished, but is also used to note the state value *during* the run of the program. For UTCP, we have to relax the interpretation for s as well. Now it too can represent the value of state *during* program execution. The only constraint is that the s' observation cannot be from a point in the execution that precedes the s observation. Our semantics admits arbitrary interleaving precisely because it relies on this interpretation.

A special observation *wait* is used in the standard UTP theories of Reactive systems to distinguish between during and after, leading to Foster's concept of Reactive Designs [15]. However, we don't need the *wait* observation because we will use the label-sets (ls, ls') to record that information.

In particular, semantic skip ($I\!I$) can be interpreted as talking about a single instant in the execution timeline, when nothing can changes, or an interval of time in which nothing changes—a so-called "stuttering" step. Stuttering statements are key for concurrency semantics to enable interference to be modelled [24,2,3].

Atomic Actions An atomic action (a) is simply a global state transformer whose effects, once started, occur immediately and completely, without any external interference. It is a relational predicate that only mentions s and s'. We use the static observables in and out to explicitly indicate the enabling label (in)

for the action, and a disabling, or "done" label *out* for the same action. We use ls and ls' to record the set of enabled labels both before and after the atomic action has run. We can define a predicate that captures the basic behaviour of such "flow-controlled" atomic action:

$$in \in ls \land a \land (ls' = (ls \setminus \{in\}) \cup \{out\}) \tag{7}$$

If in is not in ls, or predicate a is not satisfied by the current value of s, then the semantic predicate reduces to **false**. This denotes an action that is currently not enabled, and is relying on a concurrent action elsewhere to modify ls and/or s. This is analagous to a reactive design when *wait* is true.

We start by defining the notion of an eXtended basic action, where we explicitly identify the enabling labels (E), those that we remove (R), and those are newly added when done (N):

$$X(E|a|R|N) \mathrel{\widehat{=}} E \subseteq ls \land a \land ls' = (ls \setminus R) \cup N \quad \text{«·X-def·»}$$

A basic action is defined as one where E and R coincide, and is the definition used in the semantics for atomic actions

$$A(E|a|N) \mathrel{\widehat{=}} X(E|a|E|N) \quad \text{«·A-def·»}$$

The reason for the distinction is that basic actions are not closed under semantic sequential composition, whereas extended actions are:

$$\begin{array}{l} X(E_1|a|R_1|N_1); X(E_2|b|R_2|N_2) \qquad \text{«·X-then-X·»} \\ = E_2 \cap (R_1 \setminus N_1) = \emptyset \\ \quad \land X(E_1 \cup (E_2 \setminus N_1) \ \mid \ a \ ; b \ \mid \ R_1 \cup R_2 \ \mid \ (N_1 \setminus R_2) \cup N_2) \end{array}$$

The condition $E_2 \cap (R_1 \setminus N_1) = \emptyset$ characterises all those cases were the second X is enabled immediately after the first X terminates (i.e., without any outside interference). The semantic sequential composition of two (extended) basic actions captures the occurrence of both actions in sequence without any intervening interference, known as a *mumbling* step. This means that the first action once enabled, must be able to enable the second one without relying on some external agent. If expression $E_2 \cap (R_1 \setminus N_1)$ is false, this indicates that it is not possible to observe those two actions in sequence, unless some other execution thread manages to add in the missing labels in-between, as an interference step. Again this is seen a key feature in concurrency semantics [24,2,3].

Healthiness UTCP healthiness is defined to ensure that all atomic actions are in some sense always available to run, should their *in* label be inserted into ls. In effect we keep them alive by inserting them inside an infinite loop. It is interesting to note that Hoare, in his paper on Unifying Semantics of Concurrent Programming [21, Sec 2.1, p142] states

> "We will assume that there is an infinite set of possible executions of the command, which can occur in different programs, in different contexts and on different occasions."

Wheels-within-Wheels: Technically we require any healthy UTCP program predicate to be equivalent to a non-deterministic choice of how many times it repeats itself, including zero, using UTP semantic sequential composition.

$$\begin{array}{rll} P^0 \mathrel{\widehat{=}} \mathit{II} & & \text{«·seq-0·»} \\ P^{i+1} \mathrel{\widehat{=}} P \mathbin{;} P^i & & \text{«·seq-i-plus-1·»} \\ \mathbf{WwW}(P) \mathrel{\widehat{=}} \bigvee_{i \in \mathbb{N}} P^i & & \text{«·WWW-as-NDC·»} \end{array}$$

We note also, that **WwW** is monotonic and idempotent.

Label-Set Invariants: In addition, the integrity of the semantics requires that, for any construct, atomic or composite, that *in* and *out never* occupy ls at the same time. In addition, for composites, no labels generated by g should occupy ls when either *in* or *out* are present. We have two kinds of invariants, both of which are concerned with the mutual disjointness, in some sense, of a collection of sets of labels.

We introduce some shorthand notations to avoid excessively long predicates and expressions. We use '|' as a separator between things meant to be disjoint, and commas to list subsets and/or set- elements that should be unioned together. So the fragment $A, b | M, N | x, Y$ is shorthand for the mutual disjointness of $A \cup \{b\}$ and $M \cup N$ and $\{x\} \cup Y$.

To assert mutual set disjointness, we use the following shorthand, where the L_i are label-sets,

$$\{L_1 | L_2 | \ldots | L_n\} \mathrel{\widehat{=}} \forall_{i,j \in 1 \ldots n} \bullet i \neq j \implies L_i \cap L_j = \emptyset \qquad \text{«·short-disj-lbl·»}$$

The first invariant we have, Disjoint Labels (DL) is simply one that asserts, for every construct, that *in*, *out* and the labels of g are all different.

$$DL \mathrel{\widehat{=}} \{in | labs(g) | out\} \qquad \text{«·Disjoint-Labels·»}$$

Note that this invariant only refers to the static observation variables. We shall simplify further by stating that in the shorthands presented here that we use just simple g to denote $labs(g)$, so DL can we written as $\{in|g|out\}$. Some language constructs strengthen this invariant further.

We also want to assert that certain sets, necessarily mutually disjoint, can never have any of their elements in the global label-set, if any element from one of the other sets is present. Again, we have a shorthand:

$$[L_1 | L_2 | \ldots | L_n] \mathrel{\widehat{=}} \forall_{i,j \in 1 \ldots n} \bullet i \neq j \implies (L_i \cap ls \neq \emptyset \implies L_j \cap ls = \emptyset) \qquad \text{«·short-lbl-exclusive·»}$$

Note here that this is a predicate regarding the contents of ls, which is not mentioned explicitly in the shorthand. The second invariant, Label Exclusivity (LE) asserts that any point in time, only elements from of one of *in*, $labs(g)$ or *out* can be present in ls (or ls') at any point in time:

$$LE \mathrel{\widehat{=}} [in|g|out] \wedge [in|g|out]' \qquad \text{«·Exclusive-Labels·»}$$

Note that $[in|g|out]'$ simply indicates that it refers to ls' rather than ls.

UTCP Healthiness: Every healthy predicate describing a shared-variable concurrent program's behaviour is of the form $\mathbf{WwW}(C)$ for some predicate C and also satisfies DL and LE.

$$\mathbf{W}(P) \mathrel{\widehat{=}} DL \wedge LE \wedge \mathbf{WwW}(P) \quad \text{«·W-def·»}$$

We will discuss key properties of $\mathbf{W}$ when we look at validating the CKA laws (Sec 3).

Command Semantics We present the full semantics of atomic commands first, then describe an important classification of expressions and substitutions, before describing the semantics of the four composite command forms.

Atomic Commands The atomic command $\langle a \rangle$ can be very simply expressed as a basic action with the addition of healthiness conditions:

$$\langle a \rangle \mathrel{\widehat{=}} \mathbf{W}(A(in|a|out))) \quad \text{«·sem:atomic·»}$$

Here we would expect that if LE holds when this action starts, i.e. when $in \in ls$ and it gets to run, that LE' should also hold, with $out \in ls'$. Some actions we will define later will set a to be ii, which means that these actions don't modify s, but instead just focus on changing ls. We refer to these as *control-flow* actions.

Grounded and Sound Given that we have a distinction between static observations (g, in, out), and dynamic ones (s, s', ls, ls') it is worth extending this distinction to expressions and substitutions. An expression or predicate over observation variables is "ground" if the only variables present are static. The DL healthiness condition is ground, but LE is not, as it refers to ls and ls'. Ground predicates K satisfy some important laws:

$$\begin{array}{rcl} K \,;\, K & = & K \\ (K \wedge P) \,;\, Q & = & K \wedge (P \,;\, Q) \;=\; P \,;\, (K \wedge Q) \\ K \wedge \mathbf{WwW}(P) & = & \mathbf{WwW}(K \wedge P) \end{array}$$

A substitution is also deemed "ground", if all the the replacement expressions (k) are ground, and the target variables are all static. A ground substitution $[\ldots, k, \ldots / in, g, out]$, herinafter denoted by γ, distributes through semantic sequential composition, both disjoint label-set notations, and $\mathbf{WwW}$. It has no effect on semantic skip.

$$\begin{array}{rcll} (P \,;\, Q)\gamma & = & P\gamma \,;\, Q\gamma & \text{«·seq-gnd-distr·»} \\ \{L_1|\ldots|L_n\}\gamma & = & \{L_1\gamma|\ldots|L_n\gamma\} & \text{«·DL-gamma-subst·»} \\ [L_1|\ldots|L_n]\gamma & = & [L_1\gamma|\ldots|L_n\gamma] & \text{«·LE-gamma-subst·»} \\ (\mathbf{WwW}(P))\gamma & = & \mathbf{WwW}(P\gamma) & \text{«·WwW-gamma-subst·»} \\ I\!I\gamma & = & I\!I & \text{«·skip-gamma·»} \end{array}$$

A ground substitution ς, of the form $[I, labs(G), O/in, g, out]$ is *sound* if predicate $\{I|labs(G)|O\}$ holds. Soundness means the substitution cannot transform a predicate satisfying DL and LE into one that does not.

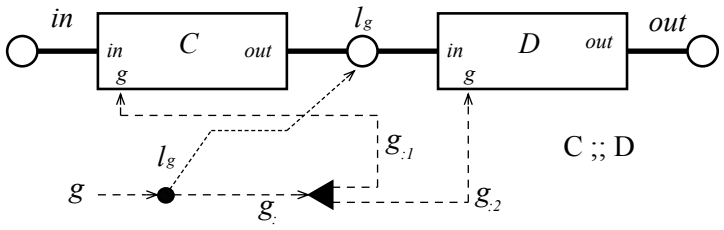

Fig. 1. Label and Generator "plumbing" for $C;;D$.

Composing Actions The semantics of composite actions basically involves using the generator to produce a suitable number of labels, that are then used in zero or more control-flow actions, of the form $A(E|ii|N)$. The left-over generator is then split as required, and then the components are "connected" into the relevant new labels and generators using sound substitutions. Finally the relevant healthiness conditions are applied. It helps to write all substitutions out in full, in order in, g, then out, so that reading them left-to-right goes with the "flow". It declutters things to then drop the part of the substitution that always reads as $/in, g, out$. So $C[\alpha, \gamma, \omega/in, g, out]$ becomes $C[\alpha, \gamma, \omega]$. This is particularly helpful during calculation.

The semantic definitions are as follows:

$$\texttt{skip} \mathrel{\widehat{=}} \langle ii \rangle$$

$$C \;;;\; D \mathrel{\widehat{=}} \mathbf{W}(C[in, g_{:1}, \ell_g] \vee D[\ell_g, g_{:2}, out]) \wedge [in|\ell_g|out]$$

$$\begin{array}{rl} C + D \mathrel{\widehat{=}} \mathbf{W}(& A(in|ii|\ell_{g1}) \vee C[\ell_{g1}, g_{1:}, out] \\ & \vee A(in|ii|\ell_{g2}) \vee D[\ell_{g2}, g_{2:}, out]\ ) \wedge [in|\ell_{g1}|\ell_{g2}|out] \end{array}$$

$$\begin{array}{rl} C \parallel D \mathrel{\widehat{=}} \mathbf{W}(& A(in|ii|\ell_{g1}, \ell_{g2}) \\ & \vee C[\ell_{g1}, g_{1::}, \ell_{g1:}] \vee D[\ell_{g2}, g_{2::}, \ell_{g2:}] \\ & \vee A(\ell_{g1:}, \ell_{g2:}|ii|out)\ ) \wedge [in|(\ell_{g1}|\ell_{g1:}), (\ell_{g2}|\ell_{g2:})|out] \end{array}$$

$$\begin{array}{rl} C^* \mathrel{\widehat{=}} \mathbf{W}(& A(in|ii|\ell_g) \\ & \vee A(\ell_g|ii|\ell_{g:}) \vee A(\ell_g|ii|out) \\ & \vee C[\ell_{g:}, g_{::}, \ell_g]\ ) \wedge [in|\ell_g|\ell_{g:}|out] \end{array}$$

First, note that all four composite definitions add further label-set invariants.

We will explain the semantics of some of the above in more detail. For sequential composition, the first component (C) uses the same in label as the construct as a whole, while the second (D) uses that same out label. The generator g is

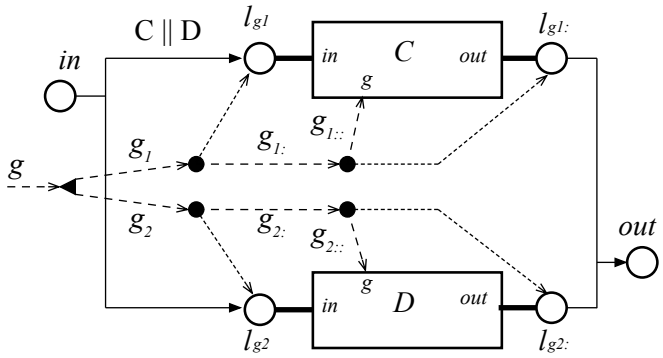

Fig. 2. Label and Generator "plumbing" for $C \parallel D$.

used to generate a label ℓ_g, that replaces *out* in C and *in* in D, effectively connecting them. The left-over generator $g_:$ is then split into two, $g_{:1}$ and $g_{:2}$, and these substituted for the g component in C and D respectively (see Figure 1).

For parallel composition (Fig. 2), we take the generator g and split it to obtain g_1 and g_2. From g_1 we generate two labels ℓ_{g1} and $\ell_{g1:}$, and leftover generator $g_{1::}$. We then use a substitution to replace all references by P to g, *in* and *out* with $g_{1::}$, ℓ_{g1} and $\ell_{g1:}$, respectively. We do something similar with g_2 and Q. We also add a top-level control-flow action that is enabled by label *in*, and adds both ℓ_{g1} and ℓ_{g2} into ls, so enabling both P and Q to start. We then have another control-flow action that waits for both of $\ell_{g1:}$ and $\ell_{g2:}$ to appear in ls, at which point they will be replaced by the top-level *out* label.

For iteration (Fig. 3), we first perform an apparently vacuous control flow action that removes label *in* from ls and replaces it with ℓ_g. That enables two actions: one that replaces ℓ_g with *out*, terminating the iteration; the other switches to label $\ell_{g:}$, which has replaced the *in* label for the body C. The *out* label of C is replaced by ℓ_g. It is important to keep the choice regarding terminate/repeat (at ℓ_g) separate from starting an execution of the body (at $\ell_{g:}$).

2.3 Formal Definition of CKA

The concept of Concurrent Kleene Algebra (CKA)was defined in a seminal paper by Hoare and collaborators [19]. This emerged as a result of ongoing work exploring the algebra and laws of shared-variable concurrency. The CKA language covers the same concepts we have in UTCP, namely atomic actions, skip, nondeterminism, sequencing, parallel composition, and iteration. It also includes the

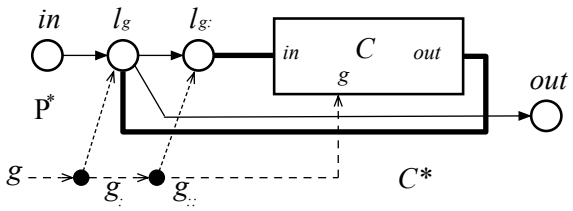

Fig. 3. Label and Generator "plumbing" for C^*.

infeasible action miracle, and a refinement ordering ($\leq$). It has since become the de-facto algebra for concurrency [19,21,23]. Using our UTCP notation we get the following set of laws, where (C,D,E,F, range over commands):

$$
\begin{array}{rcll}
C + \mathsf{miracle} & = & C & \text{«·ndc-unit·»} \\
C + C & = & C & \text{«·ndc-idem·»} \\
C + D & = & D + C & \text{«·ndc-symm·»} \\
C + (D + E) & = & (C + D) + E & \text{«·ndc-assoc·»} \\
C \mathbin{;;} \mathtt{skip} & = & C & \text{«·seq-r-unit·»} \\
\mathtt{skip} \mathbin{;;} C & = & C & \text{«·seq-l-unit·»} \\
C \mathbin{;;} (D \mathbin{;;} E) & = & (C \mathbin{;;} D) \mathbin{;;} E & \text{«·seq-assoc·»} \\
C \mathbin{;;} \mathsf{miracle} & = & \mathsf{miracle} & \text{«·seq-r-zero·»} \\
\mathsf{miracle} \mathbin{;;} C & = & \mathsf{miracle} & \text{«·seq-l-zero·»} \\
C \mathbin{;;} (D + E) & = & C \mathbin{;;} D + C \mathbin{;;} E & \text{«·seq-ndc-distr·»} \\
(C + D) \mathbin{;;} E & = & C \mathbin{;;} E + D \mathbin{;;} E & \text{«·ndc-seq-distr·»} \\
C \parallel D & = & D \parallel C & \text{«·par-symm·»} \\
C \parallel \mathtt{skip} & = & C & \text{«·par-unit·»} \\
C \parallel (D \parallel E) & = & (C \parallel D) \parallel E & \text{«·par-assoc·»} \\
C \parallel \mathsf{miracle} & = & \mathsf{miracle} & \text{«·par-zero·»} \\
C \parallel (D + E) & = & C \parallel D + C \parallel E & \text{«·par-ndc-distr·»} \\
\mathtt{skip} + C \mathbin{;;} C^* & = & C^* & \text{«·iter-fold·»} \\
C + D \mathbin{;;} E \leq E & \Longrightarrow & D^* \mathbin{;;} C \leq E & \text{«·refine1·»} \\
(C \parallel D) \mathbin{;;} (E \parallel F) & \leq & (C \mathbin{;;} E) \parallel (D \mathbin{;;} F) & \text{«·exchange·»}
\end{array}
$$

The miracle term, and the ordering in the last two laws ($\leq$) are not used in the Views paper. The precise form these take in this theory will be discussed later.

3 Methodology

The general plan for proving UTCP compliance with CKA laws, all of the form $lhs = rhs$, is to expand out their definitions, and then find a a suitable mapping

on labels between the lhs and rhs. For some laws all we need is to identify a bijection between labels, while some others need a more nuanced approach.

3.1 Composing Atomic Actions

We start by considering the general case of semantic sequential composition of two basic atomic actions:

$$
\begin{aligned}
& A(E_1|a|N_1); A(E_2|a|N_2) \\
= \ & \text{“Defn. of } A\text{”} \\
& X(E_1|a|E_1|N_1); X(E_2|a|E_2|N_2) \\
= \ & \text{“«·X-then-X·»”} \\
& E_2 \cap (E_1 \backslash N_1) = \emptyset \\
& \wedge X(\ E_1 \cup (E_2 \backslash N_1)\ \mid\ a\ ;\ b\ \mid\ E_1 \cup E_2\ \mid\ (N_1 \backslash E_2) \cup N_2\ ) \quad \text{«·A-then-A·»}
\end{aligned}
$$

Note first that this behaviour is not possible if there is an overlap between E_2 and $(E_1 \backslash N_1)$. The latter term denotes labels removed from ls when the first action runs (E_1), that are not added into ls as the action terminates (N_1). If this overlaps with E_2, then at least one enabling label is missing, and so the predicate for the second action evaluates to false, as does the whole composition—basically the first action disables the second, and hence their joint occurrence, *without interference*, can never be observed.

In [6] we described the outcome of expanding out the definition of $\langle a \rangle$, namely $\mathbf{W}(A(in|a|out))$. We find that $\mathbf{WwW}(A(in|a|out)) = \mathit{II} \vee A(in|a|out)$, because $A(in|a|out)^n = \mathbf{false}$ for $n \geq 2$. The first action replaces in by out in ls, disabling any attempt to rerun this action in the absence of outside interference.

$$\langle a \rangle = DL \wedge LE \wedge (\mathit{II} \vee A(in|a|out))$$

With different basic atomic actions we have (assuming $in_1 \neq in_2 \wedge [in_1|out_1] \wedge [in_2|out_2]$):

$$
\begin{aligned}
& A(in_1|a|out_1); A(in_2|b|out_2) \\
= \ & \text{“«·A-then-A·»”} \\
& in_2 \cap (in_1 \backslash out_1) = \emptyset \\
& \wedge X(\ in_1 \cup (in_2 \backslash out_1)\ \mid\ a\ ;\ b\ \mid\ in_1 \cup in_2\ \mid\ (out_1 \backslash in_2) \cup out_2\ ) \\
= \ & \text{“simplify, noting assumptions”} \\
& X(\ in_1 \cup (in_2 \backslash out_1)\ \mid\ a\ ;\ b\ \mid\ in_1 \cup in_2\ \mid\ (out_1 \backslash in_2) \cup out_2\ )
\end{aligned}
$$

This action is enabled if $in_1 \cup (in_2 \backslash out_1)$ is in ls. This can occur in two ways, both of which require in_1 to be present:

1. The first action is intended to enable the second ($out_1 = in_2$, let's call it mid):

$$
\begin{aligned}
& X(\ in_1 \cup (mid \backslash mid)\ \mid\ a\ ;\ b\ \mid\ in_1 \cup mid\ \mid\ (mid \backslash mid) \cup out_2\ ) \\
= \ & X(\ in_1\ \mid\ a\ ;\ b\ \mid\ in_1 \cup mid\ \mid\ out_2\ )
\end{aligned}
$$

2. The second action is independently enabled ($out_1 \neq in_2$):

$$\begin{aligned} & X(\ in_1 \cup (in_2 \backslash out_1)\ \mid\ a\ ;\ b\ \mid\ in_1 \cup in_2\ \mid\ (out_1 \backslash in_2) \cup out_2\ ) \\ = & X(\ in_1 \cup in_2\ \mid\ a\ ;\ b\ \mid\ in_1 \cup in_2\ \mid\ out_1 \cup out_2\ ) \end{aligned}$$

   We see here that the result is symmetric in the indices 1 and 2, and indeed we get same outcome if we compose the actions in the opposite order ($A(in_2|b|out_2); A(in_1|a|out_1)$). However, there is also another complication, in that some invariants are more complicated, and can have the effect that $A(in_1|a|out_1)$ in effect disables $A(in_2|b|out_2)$. We designate two atomic actions as *fully independent* if their composition is not disabled by invariants in this way. We will revisit this once we have discussed invariant preservation below.

To sum up, a basic action in our semantics is never run twice without some intervening interference step, as a consequence of the invariants. Two different actions may run back-to-back, if the first enables the second, or they are fully independent and are both currently enabled, in which case they can run back-to-back in either order.

We made use of the DL invariant (*in* and *out* are never together in ls) when showing $A(in|a|out)^2 = \mathbf{false}$. We also need to prove that atomic actions preserve the invariants. The invariants associated with atomic actions vary, depending on their role — state-change, or control-flow actions. Here are all the atomic actions explicitly mentioned in the semantics, with their additional invariants (beyond the ubiquitous DL and LE):

$$\begin{array}{rcl} A(in|a|out) : [in|out] & & A(in|ii|\ell_{g1}) : [in|\ell_{g1}|\ell_{g2}|out] \\ A(in|ii|\ell_{g2}) : [in|\ell_{g1}|\ell_{g2}|out] & & A(in|ii|\ell_g) : [in|\ell_g|\ell_{g:}|out] \\ A(\ell_g|ii|\ell_{g:}) : [in|\ell_g|\ell_{g:}|out] & & A(\ell_g|ii|out) : [in|\ell_g|\ell_{g:}|out] \\ A(in|ii|\ell_{g1}, \ell_{g2}) & : & [in|(\ell_{g1}|\ell_{g1:}), (\ell_{g2}|\ell_{g2:})|out] \\ A(\ell_{g1:}, \ell_{g2:}|ii|out) & : & [in|(\ell_{g1}|\ell_{g1:}), (\ell_{g2}|\ell_{g2:})|out] \end{array}$$

For the other actions, note that all labels mentioned in the atomic actions appear in their associated invariant. This means we need only consider the following generic case, that given that DL and LE with the appropriate substitutions for *in* and *out* hold beforehand, then after the corresponding atomic action runs, we find that the corresponding LE' holds:

$$\{E|N\} \wedge [E|N] \wedge A(E|a|N) \implies [E|N]'$$

We assume $\{E|N\} \wedge [E|N]$ and expand it to

$$(E \cap N = \emptyset) \wedge (E \cap ls \neq \emptyset \implies N \cap ls = \emptyset) \wedge (N \cap ls \neq \emptyset \implies E \cap ls = \emptyset)$$

We now evaluate the outcome of running $A(E|a|N)$:

$$A(E|a|N) = E \subseteq ls \wedge a \wedge ls' = (ls \backslash E) \cup N$$

We have two cases. The first is when either $E \not\subseteq ls$ or a is currently **false**, which means the action doesn't run so invariant preservation is vacuously satisfied. The second is when $E \subseteq ls \wedge a$ is **true**, in which case the invariant $\{E|N\} \wedge [E|N]$ indicates that no part of N appears in ls. So ls' has all of E removed and all of N added, so we can say that $N \subseteq ls' \wedge E \cap ls' = \emptyset$.

When might two distinct atomic actions *not* be fully independent? Consider the following simple example:

$$
\begin{aligned}
&\langle a \rangle + \langle b \rangle \\
&= \text{“Expand semantics of } \langle\rangle\text{, ignoring } \mathit{II}\text{, } \mathit{DL}\text{, and } \mathit{LE}\text{”} \\
&\quad A(in|a|out) + A(in|b|out) \\
&= \text{“Semantics of } +\text{”} \\
&\quad \mathbf{W}(\quad A(in|ii|\ell_{g1}) \vee A(in|a|out)[\ell_{g1}, g_{1:}, out] \\
&\qquad \vee A(in|ii|\ell_{g2}) \vee A(in|b|out)[\ell_{g2}, g_{2:}, out] \ ) \wedge [in|\ell_{g1}|\ell_{g2}|out] \\
&= \text{“Substitution”} \\
&\quad \mathbf{W}(\quad A(in|ii|\ell_{g1}) \vee A(\ell_{g1}|a|out) \\
&\qquad \vee A(in|ii|\ell_{g2}) \vee A(\ell_{g2}|b|out) \ ) \wedge [in|\ell_{g1}|\ell_{g2}|out]
\end{aligned}
$$

Recall that the sequential composition of $A(in_1|a|out_1); A(in_2|b|out_2)$ is enabled if $in_1 \cup (in_2 \backslash out_1) \subseteq ls$. Now, in our semantics above, $A(in|ii|\ell_{g1})$ and $A(\ell_{g2}|b|out)$ are independent. For $A(in|ii|\ell_{g1}); A(\ell_{g2}|b|out)$ to be enabled, we require:

$$
\begin{aligned}
&in \cup (\ell_{g2} \backslash \ell_{g1}) \subseteq ls \\
&= \text{“simplify”} \\
&\quad in \cup \ell_{g2} \subseteq ls
\end{aligned}
$$

The LE invariant permits this to be true, but the additional invariant ($[in|\ell_{g1}|\ell_{g2}|out]$) forbids this, as only one of in, out, ℓ_{g1}, or ℓ_{g2} can be present in ls at any one time. So this composition is never enabled. The use of invariants in this way in the semantics is absolutely crucial to ensuring that the semantics gives the right outcomes.

Given that we have argued that $A(E|a|N)$ preserves invariants, how do we extend this result to extended actions $X(E|a|R|N)$? First, note that the semantics only define A actions, and that all the X actions are ultimately the result of the composition of A actions. But, we know that A actions preserve the invariant, so their composition will, so hence all X actions also preserve the invariants.

3.2 W(...): the Wash Cycle

The definition of $\mathbf{W}(C)$ expands to a term of the form:

$$GI \wedge (\mathit{II} \vee C \vee (C;C) \vee (C;C;C) \vee \ldots) \wedge CI$$

Here GI (General Invariant) is just a shorthand for $DL \wedge LE$, while CI (Composite Invariant) denotes any composition-specific invariant. If C is atomic, then

$C;C = \mathbf{false}$ by construction and we are left with $II \vee C$, as explained in Sec.3.1. We need to keep in mind that we have semantic sequential composition here. If C is not atomic, then it is composed of a disjunction of control atomic actions plus one or more sub-components. Each sub-component is itself of the form $(GI \wedge \mathbf{WwW}(C_s) \wedge CI_s)\gamma$, where γ is a sound ground substitution. An immediate issue that arises is what does the semantic sequential composition $C;C$ mean precisely, when C is composite. For atomic actions it means that the second instance starts immediately after the first one ends with no other action interfering in between. This doesn't make sense as is for arbitrary composite C that itself may contain many interfering actions. What it does mean is than when the first instance of C terminates, through the running of a action that ends it, that the starting action of the second instance runs immediately. This all seems complicated, but there is a way to get to grips with the outcomes in an intuitive manner.

We start with the case where all the sub-components are themselves atomic, being a mixture of control actions of the form $A(E|ii|N)$, and state actions of the form $A(E|a|N)$. We have $C = A_1 \vee \cdots \vee A_n$, and hence, $\mathbf{W}(C)$ expands to:

$$GI \wedge (II \vee (A_1 \vee \cdots \vee A_n) \vee (A_1 \vee \cdots \vee A_n)^2 \vee \dots) \wedge CI$$

So, we can observe II, and a run of any single action A_i. We will also see runs of the form $A_i;A_j$ where $i \neq j$ and A_i does nothing that disables A_j. We also get similar terms of the form $A_i;A_j;A_k$ where $i \neq j \wedge j \neq k$. With every composite we look at, with only atomic sub-components, we find that there is a finite limit index L, such that $(A_1 \vee \cdots \vee A_n)^L = \mathbf{false}$. What we obtain is a disjunction that has every possible feasible sequential *semantic* composition of n actions, where $n \in 0 \dots L-1$ (remember that $A^0 = II$).

It is instructive to see what these look like. For sequential composition we can observe the execution of the individual actions, as well as their composition (here we use ab (ba) as shorthand for $a;b$ ($b;a$)):

$$\begin{array}{l} \langle a \rangle \;;;\; \langle b \rangle \qquad \text{«·seq-actions·»} \\ A^0 : II \vee \\ A^1 : A(in|a|\ell_g) \vee A(\ell_g|b|out) \vee \\ A^2 : A(in|ab|out) \end{array}$$

For non-deterministic choice, we also observe the control actions that implement the non-determinism:

$$\begin{array}{l} \langle a \rangle + \langle b \rangle \qquad \text{«·ndc-actions·»} \\ A^0 : II \vee \\ A^1 : A(in|ii|\ell_{g1}) \vee A(in|ii|\ell_{g2}) \vee A(\ell_{g1}|a|out) \vee A(\ell_{g2}|b|out) \vee \\ A^2 : A(in|a|out) \vee A(in|b|out) \end{array}$$

Parallelism is more complicated because both branches get enabled:

$$
\begin{array}{ll}
 & \langle a \rangle \parallel \langle b \rangle \qquad \text{«·par-actions·»} \\
A^0 : & \mathit{II} \vee \\
A^1 : & A(in|ii|\ell_{g1}, \ell_{g2}) \vee A(\ell_{g1}|a|\ell_{g1:}) \vee A(\ell_{g2}|b|\ell_{g2:}) \vee A(\ell_{g1:}, \ell_{g2:}|ii|out) \vee \\
A^2 : & A(in|a|\ell_{g1:}, \ell_{g2}) \vee A(in|b|\ell_{g2:}, \ell_{g1}) \vee \\
 & A(\ell_{g2:}, \ell_{g1}|a|out) \vee A(\ell_{g1:}, \ell_{g2}|b|out) \vee \\
 & A(\ell_{g1}, \ell_{g2}|ba|\ell_{g1:}, \ell_{g2:}) \vee A(\ell_{g1}, \ell_{g2}|ab|\ell_{g1:}, \ell_{g2:}) \vee \\
A^3 : & A(in|ba|\ell_{g1:}, \ell_{g2:}) \vee A(in|ab|\ell_{g1:}, \ell_{g2:}) \vee \\
 & A(\ell_{g1}, \ell_{g2}|ba|out) \vee A(\ell_{g1}, \ell_{g2}|ab|out) \vee \\
A^4 : & A(in|ba|out) \vee A(in|ab|out)
\end{array}
$$

We note that the full behaviour is made up of 4 actions. The first control action splits in into ℓ_{g1} and ℓ_{g2}, to enable both a and b state actions, which then run independently in either order. A final control action then waits for $\ell_{g1:}$ and $\ell_{g2:}$ to be present, and replaces them with out.

A key feature of all of the above is that there are actions of the form $A(in|\ldots|out)$ that denote a complete sequencing of atomic control and state actions that are enabled by label in, and finish with label out being deposited in ls. Another key point to take away here, is that every composite ultimately gets reduced down to a disjunction of compositions of atomic control and state change actions.

3.3 Atomic Action Flows

We use the term "flow" to describe a feasible sequential composition of atomic control and state actions. A key aspect of flow is that every well-formed command $C(in, g, out)$ has a flow that starts with $in \in ls$, then executes sub-commands using labels drawn from g, and finishes with $out \in ls'$. We call such a flow "complete". The atomic actions in «·seq-actions·», «·ndc-actions·», and «·par-actions·» denote all the possible flows for their respective constructs. The complete flows for parallel are $A(in|ab|out)$ and $A(in|ba|out)$. We are interested in complete flows as they are our basis for proving the correctness of the CKA laws w.r.t our semantics.

The labels, and the method for generating them, are key to ensuring that the semantics correctly predicts all and only the possible actions of any given program. But they are a means to an end, and we want to disregard them when discussing program equivalences. We will consider two programs to be equivalent if they have the same collection of complete flows, when actual label values are ignored. We achieve this simply by hiding all label-related observations, leaving only those that involve s and s':

$$flow(C) \mathrel{\widehat{=}} \exists in, g, out, ls, ls' \bullet C$$

Note that the way we define the semantics of C means that the following suffices as a witness to simplify the flow: $ls = \{in\} \wedge ls' = \{out\}$. When we ignore labels, we end up with a set of sequences of atomic state-changes, hence recovering a

set of traces model that is typical of most other concurrency models. If we take the actions from Sec. 3.2 and convert them to flows, we get:

$$
\begin{array}{l}
flow(\langle a\rangle \mathbin{;;} \langle b\rangle) \\
= \text{“}\langle\!\langle \cdot \text{seq-actions} \cdot \rangle\!\rangle,\ flow\text{”} \\
\quad ii \vee a \vee b \vee ab \\
\langle a\rangle + \langle b\rangle \\
= \text{“}\langle\!\langle \cdot \text{ndc-actions} \cdot \rangle\!\rangle,\ flow,\ \text{simplify”} \\
\quad ii \vee a \vee b \\
\langle a\rangle \parallel \langle b\rangle \\
= \text{“}\langle\!\langle \cdot \text{par-actions} \cdot \rangle\!\rangle,\ flow,\ \text{simplify”} \\
\quad ii \vee a \vee b \vee ba \vee ab
\end{array}
$$

It is important to note that the use of $flow$ results in information loss, and in particular, we cannot compose the output of $flow$ in any meaningful way. While the precise values of labels are not important to the overall semantics, the manner in which they are used and *kept distinct* from each other is crucial.

3.4 Composite Flows

So why focus on flows at all? The answer is that, as a concept, they make it much easier to read off the complete flows of a semantic definition. All the flows shown here were discovered/validated during the developments of the UTCP semantics, which involved a lot of tedious check calculations, and led to the development of a predicate “calculator” to make things tractable [5]. For most of that time, the “correct answer” was always self-evident. Consider the part of the definition of parallel composition $C \parallel D$ under the healthiness condition **W**:

$$
\begin{array}{l}
A(in|ii|\ell_{g1}, \ell_{g2}) \vee \\
C[\ell_{g1}, g_{1::}, \ell_{g1:}] \vee D[\ell_{g2}, g_{2::}, \ell_{g2:}] \vee \\
A(\ell_{g1:}, \ell_{g2:}|ii|out)
\end{array}
$$

We can read off the flows directly by looking at the begin and end-labels. The first (control) action goes from in to both ℓ_{g1} and ℓ_{g2}. We inductively assume that all the complete flows of $C[\ldots]$ go from ℓ_{g1} to $\ell_{g1:}$, and something similar for $D[\ldots]$. The last (control) action waits for both ℓ_{g1} and ℓ_{g2} and then proceeds to out. We made an inductive assumption above regarding the complete flows of C and D, and the effect of the substitutions applied to them. These substitutions are all sound, which means they map in, out, and the labels generated by g to distinct groups in a consistent manner. This ensure that the flows are not affected.

3.5 Atomic Control Merge

A control action with an out-label that matches the in-label of a command, can be merged with that command, as a result of **W**'s mixing. A similar situation

arises if the action has an in-label that matches the out-label of a command:

$$A(\ell_1|ii|\ell_2); C[\ell_2, g_\sigma, \ell_3] = C[\ell_1, g_\sigma, \ell_3] \quad \text{«·cntl-l-merge·»}$$
$$C[\ell_1, g_\sigma, \ell_2]; A(\ell_2|ii|\ell_3) = C[\ell_1, g_\sigma, \ell_3] \quad \text{«·cntl-r-merge·»}$$

Intuitively these laws are fairly obvious, as indeed the whole design of the label generation has been to ensure that properties like this hold true.

4 Law Proofs

The approach taken to prove a law of the form $F(C, D) = G(C, D)$, where F and G construct more complex programs using C and D, is to examine the flows of each side and establish a mapping of their labels that show every complete flow in one has a counterpart in the other. When we start to do this we quickly find that the CKA laws fall into two groups:

rearrangements (-symm,-assoc,-distr) These have the same flows, labelled differently, and label bijections suffice for correctness.

collapses (-zero,-unit,-idem) The flows on the more complex side effectively collapse to something just like the flows on the simpler side. This requires an more nuanced observability argument. In particular, all the miracle laws are collapses.

4.1 Proving Rearrangments

The rearrangement laws are:

$$C + D = D + C \quad \text{«·ndc-symm·»}$$
$$C + (D + E) = (C + D) + E \quad \text{«·ndc-assoc·»}$$
$$C \mathbin{;;} (D \mathbin{;;} E) = (C \mathbin{;;} D) \mathbin{;;} E \quad \text{«·seq-assoc·»}$$
$$C \mathbin{;;} (D + E) = C \mathbin{;;} D + C \mathbin{;;} E \quad \text{«·seq-ndc-distr·»}$$
$$(C + D) \mathbin{;;} E = C \mathbin{;;} E + D \mathbin{;;} E \quad \text{«·ndc-seq-distr·»}$$
$$C \parallel D = D \parallel C \quad \text{«·par-symm·»}$$
$$C \parallel (D \parallel E) = (C \parallel D) \parallel E \quad \text{«·par-assoc·»}$$
$$C \parallel (D + E) = C \parallel D + C \parallel E \quad \text{«·par-ndc-distr·»}$$

We will illustrate this technique with the law «·seq-ndc-distr·» regarding the distributivity of ;; through +. The approach has two phases:

1. Expand both sides of the equation using the operator definitions, and then driving all substitution inwards. We also ignore the top-level use of **W**. The LE and DL invariants distribute in through **WwW** so we can invoke them when needed. Steps here involve semantic equivalent and are separated by $=$.
2. Identify the longest runs obtained by doing semantic sequential composition of actions where the output label of one is the same as the input label of another. Steps here generate flows, by a targeted partial expansion of **WwW**, and are separated by $=_{\mathbf{w}}$. In practice we can identify these directly by careful inspection.

We start with the lefthand side:

$$
\begin{array}{ll}
& C \;;;\; (D + E) \\
= & \text{“Defn. of ;; — ignoring } \mathbf{W} \text{ for now.”} \\
& C[in, g_{:1}, \ell_g] \vee (D + E)[\ell_g, g_{:2}, out] \\
= & \text{“Defn. of +”} \\
& C[in, g_{:1}, \ell_g] \vee \\
& \left(\begin{array}{l} A(in|ii|\ell_{g1}) \vee D[\ell_{g1}, g_{1:}, out] \vee \\ A(in|ii|\ell_{g2}) \vee E[\ell_{g2}, g_{2:}, out] \end{array} \right) [\ell_g, g_{:2}, out] \\
= & \text{“substitution”} \\
& C[in, g_{:1}, \ell_g] \vee \\
& (A(in|ii|\ell_{g1}) \vee D[\ell_{g1}, g_{1:}, out])[\ell_g, g_{:2}, out] \vee \\
& (A(in|ii|\ell_{g2}) \vee E[\ell_{g2}, g_{2:}, out])[\ell_g, g_{:2}, out] \\
= & \text{“substitution”} \\
& C[in, g_{:1}, \ell_g] \vee \\
& A(\ell_g|ii|\ell_{g:21}) \vee D[\ell_{g:21}, g_{:21:}, out] \vee \\
& A(\ell_g|ii|\ell_{g:22}) \vee E[\ell_{g:22}, g_{:22:}, out]
\end{array}
$$

Now the righthand side:

$$
\begin{array}{ll}
& C \;;;\; D + C \;;;\; E \\
= & \text{“defn. +”} \\
& A(in|ii|\ell_{g1}) \vee (C \;;;\; D)[\ell_{g1}, g_{1:}, out] \vee \\
& A(in|ii|\ell_{g2}) \vee (C \;;;\; E)[\ell_{g2}, g_{2:}, out] \\
= & \text{“defn. ;;”} \\
& A(in|ii|\ell_{g1}) \vee (C[in, g_{:1}, \ell_g] \vee D[\ell_g, g_{:2}, out])[\ell_{g1}, g_{1:}, out] \vee \\
& A(in|ii|\ell_{g2}) \vee (C[in, g_{:1}, \ell_g] \vee E[\ell_g, g_{:2}, out])[\ell_{g2}, g_{2:}, out] \\
= & \text{“substitution”} \\
& A(in|ii|\ell_{g1}) \vee (C[\ell_{g1}, g_{1::1}, \ell_{g1:}] \vee D[\ell_{g1:}, g_{1::2}, out]) \vee \\
& A(in|ii|\ell_{g2}) \vee (C[\ell_{g2}, g_{2::1}, \ell_{g2:}] \vee E[\ell_{g2:}, g_{2::2}, out])
\end{array}
$$

We then read off the flows, starting with label *in* and ending with label *out*. On the lefthand side we observe the following: Command C flows from *in* to ℓ_g. Two independent control actions flow from ℓ_g to either $\ell_{g:21}$ or $\ell_{g:22}$. Command D flows from $\ell_{g:21}$ to *out*. Command E flows from $\ell_{g:22}$ to *out*. The end-to-end flows are:

$$
\begin{array}{l}
C[in, g_{:1}, \ell_g]; A(\ell_g|ii|\ell_{g:21}); D[\ell_{g:21}, g_{:21:}, out] \\
C[in, g_{:1}, \ell_g]; A(\ell_g|ii|\ell_{g:22}); E[\ell_{g:22}, g_{:22:}, out]
\end{array}
$$

A similar analysis for the righthand side results in:

$$
\begin{array}{l}
A(in|ii|\ell_{g1}); C[\ell_{g1}, g_{1::1}, \ell_{g1:}]; D[\ell_{g1:}, g_{1::2}, out] \\
A(in|ii|\ell_{g2}); C[\ell_{g2}, g_{2::1}, \ell_{g2:}]; E[\ell_{g2:}, g_{2::2}, out]
\end{array}
$$

We can immediately see that the lefthand first runs C and then makes the choice, whereas the right starts with the choice and then does the relevant sequence. If we ignore the label information, we see that both sides perform either C followed by D, or C followed by E.

For more rigour, we can apply the control merge laws to the compositions of A and C above to get a lefthand side (using «·cntl-r-merge·»):

$$C[in, g_{:1}, \ell_{g:21}]; D[\ell_{g:21}, g_{:21:}, out]$$
$$C[in, g_{:1}, \ell_{g:22}]; E[\ell_{g:22}, g_{:22:}, out]$$

and righthand side (using «·cntl-l-merge·»):

$$C[in, g_{1::1}, \ell_{g1:}]; D[\ell_{g1:}, g_{1::2}, out]$$
$$C[in, g_{2::1}, \ell_{g2:}]; E[\ell_{g2:}, g_{2::2}, out]$$

So, in both cases, the choice is incorporated in C in some sense. Now we can clearly identify a bijective mapping between the labels on each side:

$$\{in \mapsto in,\ g_{:21} \mapsto g_{1:},\ g_{:22} \mapsto g_{2:},\ g_{:21} \mapsto g_{1:},\ g_{:22} \mapsto g_{2:},\ out \mapsto out\}$$

4.2 Proving Collapses

The collapsing laws (excluding miracle) are:

$$\begin{aligned} C + C &= C && \text{«·ndc-idem·»} \\ C \mathbin{;;} \texttt{skip} &= C && \text{«·seq-r-unit·»} \\ \texttt{skip} \mathbin{;;} C &= C && \text{«·seq-l-unit·»} \\ C \parallel \texttt{skip} &= C && \text{«·par-unit·»} \\ \texttt{skip} + C \mathbin{;;} C^* &= C^* && \text{«·iter-fold·»} \end{aligned}$$

We will start with the idempotency of non-determinism «·ndc-idem·». Again we expand both sides. For the righthand side we simply show it with the identity substitution, to make comparison easier.

$$\begin{aligned} LHS \quad & C + C \\ = \ & \text{“defn +, re-arrange”} \\ & A(in|ii|\ell_{g1}) \lor C[\ell_{g1}, g_{1:}, out] \lor \\ & A(in|ii|\ell_{g2}) \lor C[\ell_{g2}, g_{2:}, out] \\ =_{\mathbf{w}} \ & \text{“«·cntl-l-merge·», twice”} \\ & C[in, g_{1:}, out] \lor C[in, g_{2:}, out] \\ RHS \quad & C \\ = \ & \text{“id. substitution”} \\ & C[in, g, out] \end{aligned}$$

We have the following flows. First the lefthand side:

$$C[in, g_{1:}, out]$$
$$C[in, g_{2:}, out]$$

Next, the righthand side:

$$C[in, g, out]$$

We have the same two labels on all instances, but three different generators. However, these generators have the same pattern of use inside C, so the internal flows will be the same, once we ignore precise label values. Here there is no label bijection as g in the RHS is associated with both $g_{1:}$ and $g_{2:}$ in the LHS. What we can do here is use the idea of a bijection between label-*sets* (here $\{g\} \leftrightarrow \{g_{1:}, g_{2:}\}$). Most of the other collapse laws are very similar.

However, it is worth looking at the iteration fold law. Upto this point we have kept quiet about iteration, limiting ourselves to constructs where the process of expanding $\bigvee_{i \in \mathbb{N}} C^i$ has been shown to terminate for some value $i > 1$. Iteration by its very nature is designed to admit an unbounded degree of unrolling. This naturally raises the question of what happens when we try to expand it.

Law $\texttt{skip} + C \mathbin{;;} C^* = C^*$ «·iter-fold·».

We will start with the righthand side (we will need it for the lefthand side in any case).

$$\begin{array}{ll} & C^* \\ = & \text{“Defn } C^*\text{”} \\ & \quad A(in|ii|\ell_g) \vee A(\ell_g|ii|\ell_{g:}) \vee A(\ell_g|ii|out) \vee C[\ell_{g:}, g_{::}, \ell_g] \\ = & \text{“}\vee\text{-idem, rearrange”} \\ & \quad A(in|ii|\ell_g) \vee A(\ell_g|ii|out) \vee \\ & \quad A(in|ii|\ell_g) \vee A(\ell_g|ii|\ell_{g:}) \vee C[\ell_{g:}, g_{::}, \ell_g] \end{array}$$

Here we use the $\vee$-idem law to justify making copies of an action. We stop at this point because, while we have one complete flow from in to out, we also have part of a flow that goes from ℓ_g to ℓ_g via C. This is by design as this is a looping construct. When label $\ell_g \in ls$, there is a choice between exiting, or executing the loop body C. We can summarise all the possible flows as:

$$A(in|ii|\ell_g); (\ A(\ell_g|ii|\ell_{g:}); C[\ell_{g:}, g_{::}, \ell_g]\ )^i ; A(\ell_g|ii|out), \qquad i \in \mathbb{N}$$

Applying «·cntl-l-merge·»to the above results in:

$$A(in|ii|\ell_g); (C[\ell_g, g_{::}, \ell_g])^i ; A(\ell_g|ii|out), \qquad i \in \mathbb{N}$$

It might appear that $C[\ell_g, g_{::}, \ell_g]$ might violate some invariants, particularly if C is atomic, so we have $A(\ell_g|a|\ell_g)$. This is not the case—the invariants don't

force the contents of ls to change, they just require that certain labels cannot be present in ls at the same time.

Finally, if we assume inductively that any expansion of C has a finite number of actions, then we can say the same for iteration (even if C itself contains an iteration).

We now turn to the lefthand side:

$$
\begin{aligned}
&\texttt{skip} + C \;;; C^* \\
&= \text{“Defn +”} \\
&\quad A(in|ii|\ell_{g1}) \vee \texttt{skip}[\ell_{g1}, g_{1:}, out] \vee \\
&\quad A(in|ii|\ell_{g2}) \vee (C \;;; C^*)[\ell_{g2}, g_{2:}, out] \\
&= \text{“Defn \texttt{skip} and ;;”} \\
&\quad A(in|ii|\ell_{g1}) \vee A(in|ii|out)[\ell_{g1}, g_{1:}, out] \vee \\
&\quad A(in|ii|\ell_{g2}) \vee \\
&\quad (C[in, g_{:1}, \ell_g] \vee C^*[\ell_g, g_{:2}, out])[\ell_{g2}, g_{2:}, out] \\
&= \text{“substitution”} \\
&\quad A(in|ii|\ell_{g1}) \vee A(\ell_{g1}|ii|out) \vee \\
&\quad A(in|ii|\ell_{g2}) \vee \\
&\quad C[\ell_{g2}, g_{2::1}, \ell_{g2:}] \vee C^*[\ell_{g2:}, g_{2::2}, out] \\
&= \text{“Defn. } C^*\text{”} \\
&\quad A(in|ii|\ell_{g1}) \vee A(\ell_{g1}|ii|out) \vee \\
&\quad A(in|ii|\ell_{g2}) \vee \\
&\quad C[\ell_{g2}, g_{2::1}, \ell_{g2:}] \vee \\
&\quad (A(in|ii|\ell_g) \vee A(\ell_g|ii|\ell_{g:}) \vee A(\ell_g|ii|out) \vee C[\ell_{g:}, g_{:}, \ell_g])[\ell_{g2:}, g_{2::2}, out] \\
&= \text{“substitution”} \\
&\quad A(in|ii|\ell_{g1}) \vee A(\ell_{g1}|ii|out) \vee \\
&\quad A(in|ii|\ell_{g2}) \vee \\
&\quad C[\ell_{g2}, g_{2::1}, \ell_{g2:}] \vee \\
&\quad A(\ell_{g2:}|ii|\ell_{g2::2}) \vee A(\ell_{g2::2}|ii|\ell_{g2::2:}) \vee A(\ell_{g2::2}|ii|out) \vee C[\ell_{g2::2:}, g_{2::2::}, \ell_{g2::2}]
\end{aligned}
$$

A key feature to note here are two components that are self-loops. One is a control action ($A(\ell_{g2::2}|ii|\ell_{g2::2:})$), while the other involves an instance of C. The first is functionally the same as II, and so is actually harmless as we can always merge it with an action that makes progress. The second ($C[\ell_{g2::2:}, g_{2::2::}, \ell_{g2::2}]$) is the occurrence of C^* on the lefthand side reduced to a self-loop. We can now

identify the following complete flows

$$A(in|ii|\ell_{g1}); A(\ell_{g1}|ii|out)$$
$$A(in|ii|\ell_{g2}); C[\ell_{g2}, g_{2::1}, \ell_{g2:}]; A(\ell_{g2:}|ii|\ell_{g2::2}); A(\ell_{g2::2}|ii|out)$$
$$A(in|ii|\ell_{g2}); C[\ell_{g2}, g_{2::1}, \ell_{g2:}]; A(\ell_{g2:}|ii|\ell_{g2::2}); A(\ell_{g2::2}|ii|\ell_{g2::2:}); A(\ell_{g2::2}|ii|out)$$
$$A(in|ii|\ell_{g2}); C[\ell_{g2}, g_{2::1}, \ell_{g2:}]; A(\ell_{g2:}|ii|\ell_{g2::2}); C[\ell_{g2::2:}, g_{2::2::}, \ell_{g2::2}]; A(\ell_{g2::2}|ii|out)$$
$$A(in|ii|\ell_{g2}); C[\ell_{g2}, g_{2::1}, \ell_{g2:}]; A(\ell_{g2:}|ii|\ell_{g2::2}); (C[\ell_{g2::2:}, g_{2::2::}, \ell_{g2::2}])^i; A(\ell_{g2::2}|ii|out)$$

Simplify these by applying control merge liberally:

$$A(in|ii|out)$$
$$C[in, g_{2::1}, out]$$
$$C[in, g_{2::1}, \ell_{g2::2}]; C[\ell_{g2::2:}, g_{2::2::}, out];$$
$$C[in, g_{2::1}, \ell_{g2::2}]; C[\ell_{g2::2:}, g_{2::2::}, \ell_{g2::2}]; C[\ell_{g2::2:}, g_{2::2::}, out];$$
$$C[in, g_{2::1}, \ell_{g2::2}]; C[\ell_{g2::2:}, g_{2::2::}, \ell_{g2::2}]; C[\ell_{g2::2:}, g_{2::2::}, \ell_{g2::2}]; C[\ell_{g2::2:}, g_{2::2::}, out];$$
$$C[in, g_{2::1}, \ell_{g2::2}]; (C[\ell_{g2::2:}, g_{2::2::}, \ell_{g2::2}])^i; A(\ell_{g2::2}|ii|out), \quad i \in 4\ldots$$

Applying control merge to the righthand side results in:

$$A(in|ii|out)$$
$$C[in, g_{::}, out]$$
$$C[in, g_{::}, \ell_g]; C[\ell_g, g_{::}, out]$$
$$C[in, g_{::}, \ell_g]; C[\ell_g, g_{::}, \ell_g]; C[\ell_g, g_{::}, out];$$
$$A(in|ii|\ell_g); (C[\ell_g, g_{::}, \ell_g])^i; A(\ell_g|ii|out) \quad i \in 4\ldots$$

Here we have the following bijection:

$$\{in \mapsto in\ , \ell_g \mapsto \ell_{g2::2:}\ , out \mapsto out\}$$

4.3 Laws involving miracle

The miracle laws are:

$$C + \mathsf{miracle} = C \qquad \text{«ndc-unit»}$$
$$C \;;;\; \mathsf{miracle} = \mathsf{miracle} \qquad \text{«seq-r-zero»}$$
$$\mathsf{miracle} \;;;\; C = \mathsf{miracle} \qquad \text{«seq-l-zero»}$$
$$C \parallel \mathsf{miracle} = \mathsf{miracle} \qquad \text{«par-zero»}$$

The real issue here is how to define miracle. The obvious thing to try is that it is simply $\mathbf{W}(\mathbf{false})$, but this simplifies to $DL \wedge LE \wedge II$ We also note that $A(in|\mathbf{false}|out) = \mathbf{false}$, and $\mathbf{W}(\mathbf{false}) = \mathbf{W}(II)$. The issue here is very reminiscent of the end of Chapter 2 of the UTP Book [17, pp55-62], where a semantics for a simple imperative language has been described. However, the loop semantics is problematic, particularly w.r.t the semantics of an infinite

loop. The issue is resolved in Chapter 3 with the concept of Designs, and in particular the new special stability observables ok and ok'. It turns out that the solution is to adopt designs at the *atomic action* level. We need to add ok,ok' as observables and to define the actions as designs:

$$A(E|a, B|N) \widehat{=} E \in ls \wedge a \vdash B \wedge ls' = (ls \setminus E) \cup N$$

We then define miracle as:

$$\mathsf{miracle} \widehat{=} A(in|\mathbf{true}, \mathbf{false}|out)$$

A simple calculation reduces this to:

$$\mathsf{miracle} = in \in ls \implies \neg ok$$

A sketch of the proof methodology follows, using the «·ndc-unit·» law as an example:

$$C + \mathsf{miracle} = C$$

We expand the LHS:

$$\begin{array}{ll} LHS & C + \mathsf{miracle} \\ & = \text{“Defn. of +”} \\ & \quad A(in|ii|\ell_{g1}) \vee C[\ell_{g1}, g_{1:}, out] \vee \\ & \quad A(in|ii|\ell_{g2}) \vee \mathsf{miracle}[\ell_{g2}, g_{2:}, out] \\ & = \text{“expand miracle”} \\ & \quad A(in|ii|\ell_{g1}) \vee C[\ell_{g1}, g_{1:}, out] \vee \\ & \quad A(in|ii|\ell_{g2}) \vee (in \in ls \implies \neg ok)[\ell_{g2}, g_{2:}, out] \\ & = \text{“substitution”} \\ & \quad A(in|ii|\ell_{g1}) \vee C[\ell_{g1}, g_{1:}, out] \vee \\ & \quad A(in|ii|\ell_{g2}) \vee (\ell_{g2} \in ls \implies \neg ok) \end{array}$$

The flows we get are:

$$C[in, g_{1:}, out] \vee (in \in ls \implies \neg ok)$$

We can easily show that for any label ℓ:

$$(\ell \in ls \wedge P \vdash Q) \vee (\ell \in ls \implies \neg ok) = (\ell \in ls \wedge P \vdash Q)$$

This means that the miracle term here is absorbed by the actions in C enabled by in.

4.4 Laws involving Refinement

The refinement laws are:

$$\begin{array}{rcll} C + D \mathbin{;;} E \leq E & \implies & D^* \mathbin{;;} C \leq E & \text{«·refine1·»} \\ (C \parallel D) \mathbin{;;} (E \parallel F) & \leq & (C \mathbin{;;} E) \parallel (D \mathbin{;;} F) & \text{«·exchange·»} \end{array}$$

The interpretation of $lhs \leq rhs$ is that it is true if every behaviour in the lhs is also a behaviour in the rhs. For us, each behaviour is a flow, partial or complete, which is defined using disjunctions. We expect that the standard definition of refinement in UTP will work here, so that $lhs \leq rhs$ simply becomes $lhs \sqsupseteq rhs$. We leave the verification of these two laws to future work.

5 Related Work

Key work was done on concurrent semantics in the 80s and 90s, with a strong focus on fully abstract denotational semantics. Notable work form this period includes that by Stephen Brookes [3] and Frank de Boer and colleagues [2]. Both looked at denotations based on the notion of sets of transition traces, these being sequences of pairs of before-after states. In order to get compositionality the traces of any program fragment had to have arbitrary "stuttering" and "mumbling" state-pairs added to capture the notion of outside interference. Full abstraction meant that the semantics had to identify programs like $skip \mathbin{;;} skip$ with $skip$, while distinguishing between $x := 2$ and $x := 1 \mathbin{;;} x := x + 1$.

The first UTP theory in this area was presented in the UTPP paper by Woodcock and Hughes [30]. This combined guarded commands [13] with the idea of action systems [1], interpreted in UTP as non-deterministic choice over guarded atomic actions, where disabled actions behave like the unit for that choice. This basic lattice-theoretic architecture for the UTPP semantics forms the foundation and inspiration for the UTCP semantics presented here.

More recently, also inspired by [14], the "UTP Views" paper by van Staden [27], starts algebraically, looking at Kleene algebras over languages. Languages here are sets of strings over an alphabet A. He then takes $A = \Sigma \times \Sigma$, which in effect encodes the Brookes model [3]. His semantics fits with the usual UTP approach to concurrency, in that it is based on traces as sequences of some notion of event.

There is however a semantics for shared-variable concurrency that is close in form to the one developed in this chapter. This is the "actions with axioms" approach of Lamport [24]. In this, the semantics of each language construct is given by a set of axioms, that are temporal logic predicates over both program variables, and additional "auxiliary" variables that manage flow of control. The meaning of a composite is given by taking the axioms that describe each of its components, and combining them with appropriate renamings. This requires being able to identify specific sub-components of any given component, and a syntactical method for doing this is described.

The recent work on Reactive Designs/Systems in UTP, by Foster and colleagues [15] is highly relevant to our work with UTCP. Here the observations ok

and *wait* are used to identify *three* phases of the operation of a reactive process: pre-, peri-, and post-condition. A pericondition is what must be true when a process is waiting for external events. Such a reactive design $[P_1 \vdash P_2|P_3]$ is a useful notation for the underlying definition $\mathbf{R}(ok \wedge P_1 \implies (P_2 \lhd wait' \rhd P_3))$. In addition there is a notion of abstract traces, and one possible use for these is to model global mutable state changes. Our work in UTCP is a converse view in a number of senses. We don't need the *wait* observable because for atomic action $A(E|a|N)$ the normal state is waiting ($wait_E = E \not\subseteq ls$) for something else to populate ls with E. In effect there is a *wait* predicate tailored for every atomic action. In addition, we have explicit global mutual state, but this could be used to model traces as histories that are only extended.

6 Conclusions

We have presented an argument that that supports our thesis that Unifying Theories of Concurrent Programming (UTCP) describes a language that satisfied the laws of Concurrent Kleene Algebra. Parts of the argument are supported by formal proof segments, while other parts are based on careful reasoned arguments and references to hopefully intuitive ideas.

6.1 Future Work

The first key issue for future work is to complete relevant proof methodologies for both miracle and refinement.

We also need to make use of the proof calculator tool support we developed [5] to check our calculations here, and we should note that proving key properties of refinement may also benefit from such support.

The original motivation for UTCP was to follow the lead of the Views paper, and to build methods from that paper, such as Rely-Guarantee [22] on top. However, it now makes more sense to layer Designs, and Reactive systems, including *Circus* on top. One of our longer-term goals is based on being able to give formal semantics to the whole chain of languages we used when using Promela/SPIN to do test generation for the real-time operating system RTEMS [10]. There we identified already existing UTP results we could use with UTCP, such as Sheng's semantics for MDESL [26], or the paper with Woodcock, Foster and myself on Heterogenous Theories [29].

Acknowledgements This work was supported, in part, by Science Foundation Ireland grants 10/CE/I1855 and 13/RC/2094 to Lero — the Irish Software Engineering Research Centre (www.lero.ie), and by ESA Contract No. `4000125572/18NL/GLC/as`, and assistance from RTEMS community.

References

1. Back, R.J.R., Kurki-Suonio, R.: Decentralization of process nets with centralized control. In: Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium

on Principles of Distributed Computing. pp. 131–142. Montreal, Quebec, Canada (17–19 Aug 1983)
2. de Boer, F.S., Kok, J.N., Palamidessi, C., Rutten, J.J.M.M.: The failure of failures in a paradigm for asynchronous communication. In: Baeten, J.C.M., Groote, J.F. (eds.) CONCUR '91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991, Proceedings. Lecture Notes in Computer Science, vol. 527, pp. 111–126. Springer (1991). https://doi.org/10.1007/3-540-54430-5_84, http://dx.doi.org/10.1007/3-540-54430-5_84
3. Brookes, S.D.: Full abstraction for a shared-variable parallel language. Inf. Comput. **127**(2), 145–163 (1996). https://doi.org/10.1006/inco.1996.0056, http://dx.doi.org/10.1006/inco.1996.0056
4. Butterfield, A.: A VDM study of fault-tolerant stable storage - towards a computer engineering mathematics. In: Woodcock, J., Larsen, P.G. (eds.) FME '93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe, Odense, Denmark, April 19-23, 1993, Proceedings. Lecture Notes in Computer Science, vol. 670, pp. 216–234. Springer (1993). https://doi.org/10.1007/BFB0024648, https://doi.org/10.1007/BFb0024648
5. Butterfield, A.: UTPCalc - A Calculator for UTP Predicates. In: Bowen, J.P., Zhu, H. (eds.) Unifying Theories of Programming - 6th International Symposium, UTP 2016, Reykjavik, Iceland, June 4-5, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10134, pp. 197–216. Springer (2016). https://doi.org/10.1007/978-3-319-52228-9_10, https://doi.org/10.1007/978-3-319-52228-9_10
6. Butterfield, A.: UTCP: compositional semantics for shared-variable concurrency. In: da Costa Cavalheiro, S.A., Fiadeiro, J.L. (eds.) Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10623, pp. 253–270. Springer (2017). https://doi.org/10.1007/978-3-319-70848-5_16, https://doi.org/10.1007/978-3-319-70848-5_16
7. Butterfield, A., Freitas, L., Woodcock, J.: Mechanising a formal model of flash memory. Sci. Comput. Program. **74**(4), 219–237 (2009). https://doi.org/10.1016/J.SCICO.2008.09.014, https://doi.org/10.1016/j.scico.2008.09.014
8. Butterfield, A., Mjeda, A., Noll, J.: UTP semantics for shared-state, concurrent, context-sensitive process models. In: 10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, July 17-19, 2016. pp. 93–100. IEEE Computer Society (2016). https://doi.org/10.1109/TASE.2016.22, https://doi.org/10.1109/TASE.2016.22
9. Butterfield, A., Sherif, A., Woodcock, J.: Slotted-circus. In: Davies, J., Gibbons, J. (eds.) Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4591, pp. 75–97. Springer (2007). https://doi.org/10.1007/978-3-540-73210-5_5, https://doi.org/10.1007/978-3-540-73210-5_5
10. Butterfield, A., Tuong, F.: Applying formal verification to an open-source real-time operating system. In: Bowen, J.P., Li, Q., Xu, Q. (eds.) Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday. Lecture Notes in Computer Science, vol. 14080, pp. 348–366. Springer (2023). https://doi.org/10.1007/978-3-031-40436-8_13, https://doi.org/10.1007/978-3-031-40436-8_13
11. Butterfield, A., Woodcock, J.: Semantic domains for handel-c. In: Flynn, S., Hurley, T., an Airchinnigh, M.M., Madden, N., McGettrick, M., Schellekens, M.P., Seda, A.K. (eds.) Second Irish Conference on the Mathematical Foundations of Computer

Science and Information Technology, MFCSIT 2002, Galway, Ireland, July 18-19, 2002. Electronic Notes in Theoretical Computer Science, vol. 74, pp. 1–20. Elsevier (2002). https://doi.org/10.1016/S1571-0661(04)80762-X, https://doi.org/10.1016/S1571-0661(04)80762-X

12. Butterfield, A., Woodcock, J.: prialt in handel-c: an operational semantics. Int. J. Softw. Tools Technol. Transf. **7**(3), 248–267 (2005). https://doi.org/10.1007/S10009-004-0181-6, https://doi.org/10.1007/s10009-004-0181-6
13. Dijkstra, E.W.: A Discipline of Programming. Series in Automatic Computation, Prentice-Hall, Englewood Cliffs , NJ , USA (1976)
14. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 287–300. ACM (2013)
15. Foster, S., Cavalcanti, A., Canham, S., Woodcock, J., Zeyda, F.: Unifying theories of reactive design contracts. Theor. Comput. Sci. **802**, 105–140 (2020). https://doi.org/10.1016/J.TCS.2019.09.017, https://doi.org/10.1016/j.tcs.2019.09.017
16. Freitas, L., Woodcock, J., Butterfield, A.: POSIX and the verification grand challenge: A roadmap. In: 13th International Conference on Engineering of Complex Computer Systems (ICECCS 2008), March 31 2008 - April 3 2008, Belfast, Northern Ireland. pp. 153–162. IEEE Computer Society (2008). https://doi.org/10.1109/ICECCS.2008.35, https://doi.org/10.1109/ICECCS.2008.35
17. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
18. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall International, Englewood Cliffs, NJ (1998)
19. Hoare, C.A.R., Möller, B., Struth, G., Wehrman, I.: Concurrent kleene algebra. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5710, pp. 399–414. Springer (2009). https://doi.org/10.1007/978-3-642-04081-8_27, https://doi.org/10.1007/978-3-642-04081-8_27
20. Hoare, C.A.R.T., Möller, B., Struth, G., Wehrman, I.: Concurrent kleene algebra. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009 - Concurrency Theory: 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings. pp. 399–414. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04081-8_27, https://doi.org/10.1007/978-3-642-04081-8_27
21. Hoare, T.: Unifying semantics for concurrent programming. In: Coecke, B., Ong, L., Panangaden, P. (eds.) Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky - Essays Dedicated to Samson Abramsky on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 7860, pp. 139–149. Springer (2013). https://doi.org/10.1007/978-3-642-38164-5_10, https://doi.org/10.1007/978-3-642-38164-5_10
22. Jones, C.B.: Tentative steps toward a development method for interfering programs. TOPLAS **5**(4), 596–619 (1983)
23. Kappé, T., Brunet, P., Silva, A., Zanasi, F.: Concurrent kleene algebra: Free model and completeness. In: Ahmed, A. (ed.) Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol.

10801, pp. 856–882. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_30, https://doi.org/10.1007/978-3-319-89884-1_30
24. Lamport, L.: An Axiomatic Semantics of Concurrent Programming Languages, pp. 77–122. Springer Berlin Heidelberg, Berlin, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82453-1_4, https://doi.org/10.1007/978-3-642-82453-1_4
25. Mjeda, A., Butterfield, A., Noll, J.: Business process modeling flexibility: A formal interpretation. In: Hammoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019. pp. 465–472. SciTePress (2019). https://doi.org/10.5220/0007577104670474, https://doi.org/10.5220/0007577104670474
26. Sheng, F., Zhu, H., He, J., Yang, Z., Bowen, J.P.: Theoretical and practical approaches to the denotational semantics for MDESL based on UTP. Formal Aspects Comput. **32**(2-3), 275–314 (2020). https://doi.org/10.1007/S00165-020-00513-4, https://doi.org/10.1007/s00165-020-00513-4
27. van Staden, S.: Constructing the views framework. In: Naumann, D. (ed.) Unifying Theories of Programming - 5th International Symposium, UTP 2014, Singapore, May 13, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8963, pp. 62–83. Springer (2014). https://doi.org/10.1007/978-3-319-14806-9_4, http://dx.doi.org/10.1007/978-3-319-14806-9_4
28. Woodcock, J., Cavalcanti, A.: A concurrent language for refinement. In: Butterfield, A., Strong, G., Pahl, C. (eds.) 5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, Ireland, 16-17 July 2001. Workshops in Computing, BCS (2001), http://ewic.bcs.org/content/ConWebDoc/4146
29. Woodcock, J., Foster, S., Butterfield, A.: Heterogeneous semantics and unifying theories. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9952, pp. 374–394 (2016). https://doi.org/10.1007/978-3-319-47166-2_26, https://doi.org/10.1007/978-3-319-47166-2_26
30. Woodcock, J., Hughes, A.P.: Unifying theories of parallel programming. In: George, C., Miao, H. (eds.) Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2495, pp. 24–37. Springer (2002). https://doi.org/10.1007/3-540-36103-0_5, http://dx.doi.org/10.1007/3-540-36103-0_5