UTCP: compositional semantics for shared-variable concurrency

Andrew Butterfield *

Lero@TCD
School of Computer Science and Statistics
Trinity College Dublin

Abstract. We present a Unifying Theories of Programming (UTP) semantics of shared variable concurrency, that is fully compositional. Previous work was based on mapping such programs, using labelling of decision points and atomic actions, to action systems, which themselves were provided with a UTP semantics. The translation to action systems was largely compositional, but their dynamic semantics was based on having all the actions collected together. Here we take a more direct approach, albeit inspired by the action-systems view, based on an abstract notion of label generation, that then exploits the standard use of substitution in UTP, to obtain a fully compostional semantics.

1 Introduction

In this paper we present a compositional UTP semantics for the shared-variable concurrent "Command" language described in the POPL13 "Views" paper[9]. We use a slightly different syntax to that in the above cited paper, where Atom is a set of atomic state-change operations,:

$$\begin{array}{l} a \in \mathsf{Atom} \\ C ::= \langle a \rangle \mid C \mathbin{;;} C \mid C + C \mid C \parallel C \mid C^* \end{array}$$

The reason for the difference is explained, in context, in Sec 4.

1.1 Motivation

The Views paper[9] presented a unification of a number of different approaches to reasoning about shared-variable concurrency, by showing how they could all be mapped onto a common base-language that they called the Command language. From a UTP perspective, we saw this as an excellent launch-pad for the kind of theory unification that is of interest to us. On a seperate note, research collaboration lead us to give a UTP semantics to a process modelling language

* 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)

called PML[1], which has the notion of basic actions that require certain resources to run, and which provide further resources as a result. Actions can be combined using sequencing, selection, branching and iteration. It turns out that PML is isomorphic to the Command language, and first steps towards a formal compositional UTP semantics of both was developed [7]. This paper moves that work on to present a compositional UTP semantics for Commands.

1.2 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[5] and Frank de Boer and colleagues[3]. 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 $skip \,;; skip$ with $skip$, while distinguishing between $x := 2$ and $x := 1 \,;; x := x + 1$.

The first UTP theory in this area was the UTPP paper[17]. This combined guarded commands[8] with the idea of action systems[2], 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 [9], the "UTP Views" paper by van Staden[15], starts algebraically, looking at Kleene alebras 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[5]. His semantics fits with the usual UTP approach to concurrency, in that it is based on traces as sequences of some notion of event. The semantics presented in this paper is based on direct relations between before- and after-program states, without any notion of traces.

2 UTP

The Unifying Theories of Programming framework [12] 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 $\mathtt{v}$ would be represented by observational variables v and v' respectively. For example, the meaning of an assignment statement might be given as follows:

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

The definition says that the assignment terminates, with the final value of variable x 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 healthiness condition can be thought of as a higher-order predicate taking a predicate as argument and returning **true** iff the predicate is healthy. Considering a healthiness predicate called "H", we might define something called **isH**, for instance. However, it considerably simplifies most UTP theories if we can find a monotonic, idempotent predicate transformer instead (**mkH**) that transforms an unhealthy predicate into a healthy one, and leaves healthy predicates unchanged. Then we can define $\mathbf{isH}(P)$ to be the assertion that tests if P is a fixed-point of **mkH**. On of the advantages of this approach is that the set of predicates generated by **mkH** inherit the complete lattice property of predicate calculus, which makes finding fixpoints for recursion a simple task. In most UTP literature the name **H** is overloaded to denote both **mkH** and **isH**. Our healthiness conditions (Sec. 6) can be expressed in this fashion.

An important characteristic of both the UTP theories referred to above, is that their predicates are interpeted as a relation between the before-state and after-state of a *complete* program execution.

The "standard" treatment of concurrency in UTP[12, Chps. 7,8], is focussed on local-state concurrency, without any mutable state variables. Here it becomes necessary to observe the program state at intermediate points in its execution, typically when the program is waiting for external events to occur. This neccesitates another pair of Boolean observations, $wait$ and $wait'$ that indicate such waiting. We do not give any further details regarding these theories, but instead mention them simply to make the observation that here the predicates are interpreted as a relation between the before-state, and some *subsequent* intermediate or final state of the complete execution.

Our focus in this introduction on how the predicates are *interpreted* in terms of program state is important, because the theory presented in this paper involves yet more adjustments in interpretation, as explained in Sections 4 and 8.

3 Labels

In the work on UTPP [17], the programs were required to have labels in certain places. Our semantic approach here is no different to the above with respect to the need for labels, but we generate them automatically in a way that supports compositional reasoning within the UTP framework.

Given some notion of labels ($l \in Lbs$), we want a notion of 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 define the following terse label and generator expression syntax:

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

Here $G_{:}$ denotes the generator left once *new* has been run on G, with ℓ_G denoting the label so generated. Expressions G_1 and G_2 denote the two outcomes of applying *split* to G. We use $labs(G)$ to denote all the labels that G can generate and we require the following laws to hold:

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

The simplest model for a generator that satisfies the above constraints is one that represents the label ℓ_G by the expression G itself. The reason for this shorthand is that without it we would have to write something like the following

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

instead of $\ell_{g1:2:}$. 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 substition "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.

4 Observations

Any UTP theory has to clearly define its *alphabet*, that is, the set of observational variables that define its domain of discourse. The theory presented here is inspired by UTPP[17] and uses some of the observations presented there: 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 theory in [17] also uses ok and ok', but these are not required here —they can added in though as derived observations, if wanted.

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. For details of how this works, see Section 7.3.

In effect we are exploiting the fact that our language is block-structured with only one entry and exit point for each construct, in order to be able to decouple the semantics of an atomic action from whatever might come next. Dealing with that is the responsibiity of the semantics of language composites.

To summarise, our semantics is built upon basic atomic state-change actions that modify global shared state s. The concurrent flow of control is managed using a global dynamic label-set ls and the static association of a label generator g and two distinguished labels in, out, with every language construct.

This brings us to an important distinction between the usual approach taken by UTP regarding the distinction between syntax and semantics. The usual approach, inspired by the slogan "programs are predicates"[10,11], is to treat syntax and semantics as the same thing. A program's syntax is simply a shorthand notation for its semantics. So, the program text `x := x+y` *is* a predicate, a shorthand for the more verbose $x' = x + y \wedge y' = y$ [1] . in particular, the notation for sequential composition, $P; Q$, is a shorthand for $\exists obs_m \bullet$

[1] Assuming `x` and `y` are the only variables

$P[obs_m/obs'] \wedge Q[obs_m/obs]$, where obs (obs') refers to all the before- (after-) observations. 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 interfence. We define it as follows:

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

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 semantic skip (II), the unit for semantic sequential composition, as

$$II \mathrel{\widehat{=}} ls' = ls \wedge s' = s \tag{7}$$

5 Atomic Actions

An atomic action (a) is simply a global state transformer whose effects, once started, occur immediately and completely, without any external interference. We can consider it be a relational predicate that only mentions s and s'. In the semantics of UTPP, and here for UTCP, every atomic action is labelled.

As already stated, we use a to denote the predicate describing the core global state-changes, and 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:

$$in \in ls \wedge a \wedge ls' = (ls \setminus \{in\}) \cup \{out\} \tag{8}$$

In short: the action when its in-label is in ls, it that it performs the state-change specified by a, and replaces the in-label by the out-label, in the updated set ls' of enabled labels. 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**.

The semantics of a running composite program, as per the action systems approach used in [17], is to imagine all of the labelled atomic actions collected into one large non-deterministic choice, itself in a loop that runs until some distinguished stop-label appears in the enabled label-set. The whole thing is initialised by enabling at least one atomic action in-label. We can imagine a predicate transformer run that takes such a non-deterministic choice (NDC) and produces a relational predicate that defines its overall running behaviour:

$$NDC = \bigvee_{j \in 0 \ldots n} (in_j \in ls \wedge a_j \wedge ls' = (ls \setminus \{in_j\}) \cup \{out_j\}) \tag{9}$$

$$run(NDC) \mathrel{\widehat{=}} ls' = \{in_0\};\ \textbf{while}\ out_n \notin ls\ \textbf{do}\ NDC \tag{10}$$

Here we assume in_0 and out_n are the initial and final labels of the whole program. In effect, the meaning of a shared-variable concurrent program is all the interleavings of atomic actions that are consistent with flow-of-control restrictions, with each interleaving being a series of atomic actions sequentially composed *semantically*, using ; as defined in Eqn. 6.

Given that we will be sequentially composing a lot of predicates like 8, we shall introduce a shorthand notation that we refer to as a "basic action", which refers to sets of labels called E (enablers) and N (new):

$$A(E \mid a \mid N) \mathrel{\widehat{=}} E \subseteq ls \wedge a \wedge ls' = (ls \setminus E) \cup N \quad \text{«A-def»}$$

The plan is to then produce some laws governing the semantic sequential compositions of basic actions ($A(E_1 \mid a_1 \mid N_1); A(E_2 \mid a_1 \mid N_2)$), but we quickly discover that in general the outcome cannot be expressed as a single instance of the form $A(E \mid a \mid N)$. Consider $A(l_1 \mid a \mid l_2); A(l_2 \mid b \mid l_3)$, in a starting state where both l_1 and l_2 are in ls. The overall result is a combined action that needs l_1 to start, and adds in l_3 at the end, but also removes both l_1 and l_2. So, in order to effectively calculate with the theory (see Sec. 8), we need to generalise the basic action idea to an eXtended basic action, where we explicitly identify the labels that we remove (R):

$$X(E \mid a \mid R \mid A) \mathrel{\widehat{=}} E \subseteq ls \wedge a \wedge ls' = (ls \setminus R) \cup A \quad \text{«X-def»}$$

Clearly $A(E \mid a \mid N) = X(E \mid a \mid E \mid N)$. We can now prove the following composition law:

$$\begin{array}{l} X(E_1 \mid a \mid R_1 \mid A_1); X(E_2 \mid b \mid R_2 \mid A_2) \qquad \text{«X-then-X»} \\ = E_2 \cap (R_1 \setminus A_1) = \emptyset \\ \quad \wedge X(E_1 \cup (E_2 \setminus A_1) \mid a \,;\, b \mid R_1 \cup R_2 \mid (A_1 \setminus R_2) \cup A_2) \end{array}$$

The condition $E_2 \cap (R_1 \setminus A_1) = \emptyset$ characterises all those cases were the second X is enabled immediately after the first X terminates (i.e., without any outside interference). This brings us to a very important aspect of how these predicates are to be interpreted. The semantic sequential composition of two basic actions captures the occurence 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. The expression $E_2 \cap (R_1 \setminus A_1)$ is all of the labels in E_2 that are removed (R_1) by the first action, but are not added back in (A_1). If this not empty then some of the labels from E_2 will not be present, and so the second action has been disabled by the first. So the whole predicate reduces to **false**, indicating that it is not possible to observe those two actions in sequence, unless some other execution thread manages to add in the missing E_2 labels inbetween.

6 Healthiness

Wheels-within-Wheels The key intuition behind this compositional semantics was to take the top-level *run* function of the action-system based semantic model used in UTPP, and drive it inwards to every level of the program. The original *run* can be defined in the context of this theory as

$$run(P) = ls := \{in\}; \neg(out \in ls) * P$$

However this failed to keep atomic components "live". They could never be re-executed, as would be required if they were within an iteration. Instead it was realised that every construct (atomic and composite) would have to be within an infinite loop.

$$\mathbf{true} * P \quad \text{«·WWW-as-loop·»}$$

This wrapping of the basic property in an infinite loop would occur at every level of the program hierarchy, hence the our use of the phrase "Wheels within Wheels" (WwW) to refer to this principle.

It should be noticed that this theory underwent a large enumber of iterations before the WwW principle was finally elucidated properly and shown to give the right results. Earlier work in this area, reported in [7], still had *run* as shown above and was not fully compositional. The number and complexity of the test calculations needed to debug, develop and validate the theory presented in this paper necessitated the develoment of a bespoke "UTP Calculator"[6].

This bold step turns out to be remarkably effective, with some quite counter-intuitive outcomes. However it does depend on a specific tweak to the definition of an atomic action. In effect we define an atomic action as placing a basic action inside such a loop, but within a non-deterministic choice between it and a *stuttering* step, denoted by UTP's *semantic* skip (II):

$$\begin{aligned} \langle a \rangle &= \mathbf{true} * (II \vee A(in \mid a \mid out)) \\ II &= s' = s \wedge ls' = ls \end{aligned}$$

A result of this is that this stuttering step gets propagated up to enclosing composites, so in effect we see $\mathbf{true} * C = \mathbf{true} * (II \vee C)$ where C is any predicate denoting the semantics of a command. Given that our loop bodies always have such a disjunction, it then becomes interesting to ask what this looks like:

$$\mathbf{true} * (II \vee P) = \bigvee_{i \in \mathbb{N}} P^i \quad \text{«·loop-as-NDC·»}$$

We find that such a loop is equivalent to a non-deterministic choice over the number of times that P is repeated, including zero.

We choose to *define* the healthiness condition as this large choice.

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

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

6.1 Label-Set Invariants

The semantics we propose here depends on the careful management of when specific labels are, or are not, present in the global label-set ls. Key to the success of this semantics is a collection of label-set invariants which characterise proper label-set contents, which are preserved by all label-set manipulations performed by our semantic definitions. 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 seperator between things meant to be disjoint, and commas to list subsets and/or set- elements that should be unioned together. So the fragment $A, b \mid M, N \mid x, Y$ is shorthand for the mutual disjointness of $A \cup \{a\}$ 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,

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

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:

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

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. [2]

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

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 $\{in \mid g \mid out\}$. We also need stronger Label Exclusivity invariants, regarding which labels can, or cannot, occur in the global label set at any one time. There is not one such invariant, but rather we have that some language constructs may define their own variation, in order to ensure that flow of control is correctly managed.

There is a general version of the invariant (LE) that holds for all language constructs that asserts that any point in time, only elements from of one of in, $labs(g)$ or out can be present in ls at any point in time:

$$LE \mathrel{\widehat{=}} [in \mid g \mid out] \quad \text{«·Exclusive-Labels·»}$$

Note that we also just use a dash to make the same assertion about the allowable contents of ls' by simply writing $[in \mid g \mid out]'$.'

[2] The theory can be developed using only g as a static observation, and letting ℓ_g and $\ell_{g:}$ play the role of in and out respectively, in which case Disjoint Labels is automatically satisfied. However, while this results in an entirely equivalent theory, it is notationally more obscure making it harder to interpret and check.

So, in summary, we have that 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 maintains LE.

7 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.

7.1 Atomic Commands

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

$$\begin{array}{rll} \mathbf{W}(P) \mathrel{\widehat{=}} DL \wedge LE \wedge \mathbf{WwW}(P) & & \text{«·W-def·»} \\ \langle a \rangle \mathrel{\widehat{=}} \mathbf{W}(A(in \mid a \mid out))) & & \text{«·sem:atomic·»} \end{array}$$

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 $out \in ls'$.

7.2 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. The reason for this is do with the fact that, by design, semantic sequantial composition ignores the static variables. An expression is "ground" if the only variables present are static. A substitution is also deemed "ground", if all the the replacement expressions are ground, and the target variables are all static. A desired consequence of this is that ground substitutions γ will distribute through semantic sequential composition, semantic skip, and both disjoint label-set notations

$$\begin{array}{rl} (P \,;\, Q)\gamma = P\gamma \,;\, Q\gamma & \text{«·seq-gnd-distr·»} \\ II\gamma = II & \text{«·skip-gamma·»} \\ \{L_1 \mid \ldots \mid L_n\}\gamma = \{L_1\gamma \mid \ldots \mid L_n\gamma\} & \text{«·DL-gamma-subst·»} \\ [L_1 \mid \ldots \mid L_n]\gamma = [L_1\gamma \mid \ldots \mid L_n\gamma] & \text{«·LE-gamma-subst·»} \end{array}$$

Groundness is not enough, we also require substitutions to be "sound" in the sense that they cannot transfrom a situation that satisfies DL or LE into one that does not. A ground substitution ς, of the form $[G, I, O/g, in, out]$ is *sound* if $\{I \mid G \mid O\}$ holds. We will see that all substitutions in the semantic definitions below are sound, and that this is easy to check by inspection.

$$
\begin{array}{rll}
P \mathbin{;;} Q \mathrel{\widehat{=}} \mathbf{W}(P[g_{:1}, \ell_g/g, out] \vee Q[g_{:2}, \ell_g/g, in]) & & \text{«·sem:seq·»} \\
P \parallel Q \mathrel{\widehat{=}} \mathbf{W}(\ A(in \mid ii \mid \ell_{g1}, \ell_{g2}) \vee & & \text{«·sem:par·»} \\
\qquad P[g_{1::}, \ell_{g1}, \ell_{g1:}/g, in, out] \vee & & \\
\qquad Q[g_{2::}, \ell_{g2}, \ell_{g2:}/g, in, out] \vee & & \\
\qquad A(\ell_{g1:}, \ell_{g2:} \mid ii \mid out)\ ) & & \\
P + Q \mathrel{\widehat{=}} [g_1 \mid g_2] \wedge \mathbf{W}(\ P[g_1/g] \vee Q[g_2/g]\ ) & & \text{«·sem:NDC·»} \\
P^* \mathrel{\widehat{=}} \mathbf{W}(\ A(in \mid ii \mid \ell_{g:}) \vee & & \text{«·sem:star·»} \\
\qquad A(\ell_{g:} \mid ii \mid \ell_g) \vee & & \\
\qquad A(\ell_{g:} \mid ii \mid out) \vee & & \\
\qquad P[g_{::}, \ell_g, \ell_{g:}/g, in, out]\ ) & &
\end{array}
$$

Fig. 1. Composite Semantics

7.3 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 $A(E \mid ii \mid N)$, where ii is atomic skip that simply asserts $s' = s$. The left-over generator is the 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 supplied. A key principle is to ensure that when any sub-component is "active", that is, at least one of its labels is present in ls, that none of the labels of the parent, other than those explicitly shared with the sub-component, are themsleves in ls. This prevents a parent starting a spurious copy of a sub-component while that sub-component is actually running. The semantic definitions are listed in Fig. 1. Careful inspection of all the substitutions present will confirm that they are all sound. Given that the invariant LE, which is $[in \mid labs(g) \mid out]$, is part of the definition of $\mathbf{W}$, then we have it statisfied, by definition, by any sub-components. From the perspective of the parent composite, this means that $LE\varsigma$ also holds, where ς ranges over all the sound substitutions used in the definition of the parent's semantics. For example, for program sequential composition, we not only assert $[in \mid g \mid out]$, but can also infer $[in \mid g_{:1} \mid \ell_g]$ and $[\ell_g \mid g_{:2} \mid out]$. Some constructs require a little more than just DL, LE and the sound substitution variants just discussed. For example in program non-deterministic choice $(+)$, we need to insist that only one branch is taken, hence the need to assert that if any label is present in ls from one component then none from the other are present.

In summary, we have have predicate semantics for atomic and composite program constructs, in which everything at every level is wrapped in an infinite loop. This seems to be completely counter-intuitive: a program that consists of a single atomic action may wait for a while while external interference rumbles on, but eventually it should get "scheduled", perform its atomic action and then effectively stop. How is this consistent with looping forever? To see the answer to this question, it helps to consider such simple examples, and this brings up the issue of *calculation*.

8 Calculations

This semantic theory was validated by a series of test calculations done to ensure that it was making the right predications about program behaviour. This typically involved taking small programs with a few atomic actions and trying to simplify their semantic predicates down to a non-deterministic choice of atomic action sequences. Some of the calculations proved to be very long, repetitive and tedious, motivating the UTP-calculator development [6].

We shall start by sketching out a test calculation for $\langle a \rangle$, where the objective is to reduce it down to a predicate involving just basic atoms.

$$\begin{array}{ll} & \langle a \rangle \\ = & \mathbf{W}(A(in \mid a \mid out))) & \text{«·sem:atomic·»} \\ = & DL \wedge LE \wedge \mathbf{WwW}(A(in \mid a \mid out)) & \text{«·W-def·»} \\ = & DL \wedge LE \wedge \bigvee_i A(in \mid a \mid out)^i & \text{«·WWW-as-NDC·»} \end{array}$$

At this point what remains is to compute $A(in \mid a \mid out)^i$ for $i \in \mathbb{N}$. The cases of $i = 0, 1$ are straightforward. Computing $i = 2$ is easy:

$$\begin{array}{ll} & A(in \mid a \mid out)\ ;\ A(in \mid a \mid out) & \text{«·X-def·»} \\ = & X(in \mid a \mid in \mid out)\ ;\ X(in \mid a \mid in \mid out) & \text{«·X-then-X·»} \\ = & \{in\} \cap (\{in\} \setminus \{out\}) = \emptyset \wedge X(\dots) & \text{set theory} \\ = & \mathbf{false} \wedge X(\dots) \end{array}$$

We see that $A(in \mid a \mid out)^2 = \mathbf{false}$, and as **false** is a zero for semantic sequential composition, we can deduce that $A(in \mid a \mid out)^i = \mathbf{false}$ for all $i \geq 2$. So our final result is

$$\langle a \rangle = DL \wedge LE \wedge (II \vee A(in \mid a \mid out)) \tag{11}$$

Ignoring the healthiness conditions, this boils down to two possible observations we can make of $\langle a \rangle$: either we observe suttering—no change in state or label-sets (II) or we see the complete execution of the underlyng basic action $A(in \mid a \mid out)$.

Test calculations for simple usage of most of the composites is essentially the same. One slight complication is that the contents of **WwW** in theses cases is a disjunction of terms, rather than a single basic action, so we first simplify these out, applying all substitutions, to get a term Q of the form ($II \vee$ *basic actions*). We need to compute Q^i for $i \geq 2$, and sequential composition distributes through disjunction, so we obtain resulting terms of the same form, by repeated application of law «·X-then-X·». A large number of these have results with the set side-condition that evaluates to **false**, as per the $i = 2$ example above—these terms vanish. There are other terms produced that do not vanish, but some of these can also be eliminated, because their enabling set violates the Label Exclusivity invariant. All remaining terms have the form $X(E \mid a \mid R \mid N)$, and some of these can be immediately re-written to $A(E \mid a \mid N)$, if $R = E$. In every test calculation we have done it turns out that the others, where $R \neq E$ can

$$
\begin{aligned}
\langle a\rangle \mathbin{;;} \langle b\rangle &= II \vee A(in \mid a \mid \ell_g) \vee A(\ell_g \mid b \mid out) \vee A(in \mid ab \mid out)\\
\langle a\rangle + \langle b\rangle &= II \vee A(in \mid ii \mid \ell_{g1}) \vee A(in \mid ii \mid \ell_{g2}) \vee A(\ell_{g1} \mid a \mid out)\\
&\quad \vee A(\ell_{g2} \mid b \mid out) \vee A(in \mid a \mid out) \vee A(in \mid b \mid out)\\
\langle a\rangle \parallel \langle b\rangle &= II \vee A(in \mid ii \mid \ell_{g1}, \ell_{g2}) \vee A(\ell_{g1:}, \ell_{g2:} \mid ii \mid out) \vee A(\ell_{g1} \mid a \mid \ell_{g1:})\\
&\quad \vee A(\ell_{g2} \mid b \mid \ell_{g2:}) \vee A(in \mid a \mid \ell_{g1:}, \ell_{g2}) \vee A(in \mid b \mid \ell_{g2:}, \ell_{g1})\\
&\quad \vee A(\ell_{g1}, \ell_{g2} \mid ba \mid \ell_{g1:}, \ell_{g2:}) \vee A(\ell_{g1}, \ell_{g2} \mid ab \mid \ell_{g1:}, \ell_{g2:})\\
&\quad \vee A(\ell_{g2:}, \ell_{g1} \mid a \mid out) \vee A(\ell_{g1:}, \ell_{g2} \mid b \mid out) \vee A(in \mid ba \mid \ell_{g1:}, \ell_{g2:})\\
&\quad \vee A(in \mid ab \mid \ell_{g1:}, \ell_{g2:}) \vee A(\ell_{g1}, \ell_{g2} \mid ba \mid out) \vee A(\ell_{g1}, \ell_{g2} \mid ab \mid out)\\
&\quad \vee A(in \mid ba \mid out) \vee A(in \mid ab \mid out)
\end{aligned}
$$

Fig. 2. Some Test Calculation Results. Here ab (ba) is short for $a; b$ ($b; a$).

also be re-written, because LE says that none of $R \setminus E$ can be present in ls when anything from E is present, so the removal of those labels is ineffective, as they are never present when that action is enabled. So, the outcome is that we get final results where every basic action can be written in the A-form. All of these aspects of these test calculations are supported by current versions of the tool described in [6]. If there is no use of the iteration construct (P^*), then all calculations terminate because there is always some i for which Q^i evaluates to **false**. Any use of the language iteration construct however results in having terms for all values of i.

Some calculation results are shown in Fig. 2. If we look at the result for $\langle a\rangle;;\langle b\rangle$ we have II, the stuttering step, and $A(in \mid ab \mid out)$ which is the complete exection of both actions without interference (mumbling), and $A(in \mid a \mid \ell_g)$ that shows the execution of a, to an intermediate point where b has yet to occur. These three observations are consistent with the idea that our predicates are relations between a starting state and some subsequent or final state. However we also have action $A(\ell_g \mid b \mid out)$, which is an observation that begins after action a has already occured, and just observes the behaviour of b alone. What has happened with this UTP theory of concurrency is that it is now no longer insists that the "before" observation is pinned to be the start of the program. Now we are able to observe program behaviour that can both start and end at what are intermediate points in the lifetime of the program.

If we look at $\langle a\rangle + \langle b\rangle$, we also explictly see the control-flow "decisions", such as $A(in \mid ii \mid \ell_{g1})$ where the decision to execute a is made. This will remove in from ls if it runs, so disabling the other choice, denoted by $A(in \mid ii \mid \ell_{g2})$. By contrast, in $\langle a\rangle \parallel \langle b\rangle$ the initially enabled action is $A(in \mid ii \mid \ell_{g1}, \ell_{g2})$, which activates both a and b. The control flow action $A(\ell_{g1:}, \ell_{g2:} \mid ii \mid out)$ delays termination until both atomic actions are done.

Finally, we stress that the explicit inclusion of labels in the final results is essential in order to ensure compositionality. In [7] we had the explicit run form, and this reduced the semantics of $\langle a\rangle$ to a, that of $\langle a\rangle + \langle b\rangle$ to $a \vee b$ and $\langle a\rangle \parallel \langle b\rangle$ to $ab \vee ba$. While this looks cleaner, it has lost too much information, and we cannot compose these further to get correct answers. With the explcitly labelled

semantics presented here for UTCP, we can, for example, correctly compute $(\langle a \rangle \mathbin{;;} \langle b \rangle) \parallel \langle c \rangle$ by replacing the first ;; term by its expansion from Fig. 2.

9 Conclusions

We have presented a compositional, denotational UTP semantics of shared-variable concurrency. It is "explicit" in the sense that it can enumerate all the observations that it is possible make of the program's own behaviour, in any time-slot.

The usefulness of this theory is that it is in a form that makes it very easy for us to link it to other UTP theories about approaches to concurrency, such as Rely-Guarantee[13,16], Owicki-Gries[14], Concurrent Seperation Logic[4], all featured in the Views paper. This is precisely because it is formulated as a before-after relation, with the twist that before and after observations can occur at any time during program execution, with the obvious proviso that "before" precedes "after".

9.1 Future Work

While careful inspection and test calculations give us a high level of confidence in the validity of our semantics, we still need to demonstrate that the algebraic laws of Concurrent Kleene Algebra[] can be derived form our semantics. We also need to show how the standard operational semantics can be recovered.

Of particular interest is to explore the connection between UTCP and rely-guarantee. Given that we can explictly provide all atomic actions and their mumblings, we see a good opportunity to explore how this can be exploited to analyse how well one or more program steps satisfy their guarantee obligation, given a reliable environment.

References

1. Atkinson, D.C., Weeks, D.C., Noll, J.: Tool support for iterative software process modeling. Information & Software Technology 49(5), 493–514 (2007), http://dx.doi.org/10.1016/j.infsof.2006.07.006
2. 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)
3. 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), http://dx.doi.org/10.1007/3-540-54430-5_84
4. Brookes, S.: A revisionist history of concurrent separation logic. Electr. Notes Theor. Comput. Sci. 276, 5–28 (2011), https://doi.org/10.1016/j.entcs.2011.09.013

5. Brookes, S.D.: Full abstraction for a shared-variable parallel language. Inf. Comput. 127(2), 145–163 (1996), http://dx.doi.org/10.1006/inco.1996.0056
6. 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
7. Butterfield, A., Mjeda, A., Noll, J.: UTP Semantics for Shared-State, Concurrent, Context-Sensitive Process Models. In: Bonsangue, M., Deng, Y. (eds.) TASE 2016 10th International Symposium on Theoretical Aspects of Software Engineering. pp. 93–100. IEEE (Jul 2016)
8. Dijkstra, E.W.: A Discipline of Programming. Series in Automatic Computation, Prentice-Hall, Englewood Cliffs , NJ , USA (1976)
9. 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)
10. Hehner, E.C.R.: Predicative programming part i & ii. Commun. ACM 27(2), 134–151 (Feb 1984)
11. Hoare, C.A.R.: Programs are predicates. In: Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages. pp. 141–155. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
12. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall International, Englewood Cliffs, NJ (1998)
13. Jones, C.B.: Developing methods for computer programs including a notion of interference. Ph.D. thesis, University of Oxford, UK (1981), http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.259064
14. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. 6, 319–340 (1976), https://doi.org/10.1007/BF00268134
15. 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), http://dx.doi.org/10.1007/978-3-319-14806-9_4
16. van Staden, S.: On rely-guarantee reasoning. In: Hinze, R., Voigtländer, J. (eds.) Mathematics of Program Construction: 12th International Conference, MPC 2015, Königswinter, Germany, June 29–July 1, 2015. Proceedings. pp. 30–49. Springer International Publishing, Cham (2015), https://doi.org/10.1007/978-3-319-19797-5_2
17. 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), http://dx.doi.org/10.1007/3-540-36103-0_5