Metadata of the chapter that will be visualized in SpringerLink

Book Title	Formal Methods: Foundations and Applications
Series Title	
Chapter Title	UTCP: Compositional Semantics for Shared-Variable Concurrency
Copyright Year	2017
Copyright HolderName	Springer International Publishing AG

Corresponding Author	Family Name	**Butterfield**
	Particle	
	Given Name	**Andrew**
	Prefix	
	Suffix	
	Division	Lero@TCD, School of Computer Science and Statistics
	Organization	Trinity College Dublin
	Address	Dublin 2, Ireland
	Email	butrfeld@tcd.ie
	ORCID	http://orcid.org/0000-0002-2337-2101

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 compositional semantics.

UTCP: Compositional Semantics for Shared-Variable Concurrency

Andrew Butterfield(✉)

Lero@TCD, School of Computer Science and Statistics,
Trinity College Dublin, Dublin 2, Ireland
butrfeld@tcd.ie

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 compositional semantics.

1 Introduction

In this paper we present a compositional semantics for a simple abstract shared-variable concurrent language, called the "Command" language presented in Fig. 1. The Command language is very simple, with sequential composition $(C_1 \mathbin{;;} C_2)$, and only non-deterministic choices, for alternative execution paths $(C_1 + C_2)$ or deciding when to terminate a loop (C^*). The parallel composition $(C_1 \parallel C_2)$ allows arbitrary interference by each side on any variables, all of which are considered here to be global and shared. The semantics we present does not itself need to deal explicitly with any shared variables, but simply assumes a shared state s and the existence of atomic state-change actions a. This Command language corresponds directly to Concurrent Kleene Algebra (CKA) [14].

Our interest in this language stems from our general work within the Unifying Theories of Programming (UTP) framework [13], in which we seek to find ways to unify the semantics of a wide range of programming and specification languages, and language features, in order to be able to reason formally about systems built using a mix of such languages. The Command language in this paper is based on that introduced in the "Views" paper [10], which describes how a range of approaches to reasoning about shared-variable concurrency can

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).

S. Cavalheiro and J. Fiadeiro (Eds.): SBMF 2017, LNCS 10623, pp. 1–18, 2017.
https://doi.org/10.1007/978-3-319-70848-5_16

$$
\begin{array}{rcll}
a & \in & \mathsf{Atom} & \text{Atomic state-change actions} \\
C & ::= & \langle a \rangle & \text{Atomic Command} \\
 & \mid & C \mathbin{;;} C & \text{Sequential Compostion} \\
 & \mid & C + C & \text{Non-deterministic Choice} \\
 & \mid & C \parallel C & \text{Parallel Compostion} \\
 & \mid & C^{*} & \text{Non-deterministic Iteration}
\end{array}
$$

Fig. 1. Command language syntax

be mapped down onto CKA, and the Command language. Approaches covered in [10] include various Separation logics [8], type-theories, Owicki-Gries [20], and Rely-Guarantee [17], among others. Our intention in developing a UTP semantics of the Command language is to be able to use it as a foundation on which to build UTP theories of the above approaches that will be easy to link together. In effect we hope to use the results of the Views paper as a conceptual architecture to organise our work.

Another independent motivation for this work is a research collaboration that led us to give a UTP semantics to a process modelling language 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. We published initial work on a UTP semantics for PML [7], noting that it is essentially the same as the Command language. The semantics we gave in [7] was not compositional, however, and finding a fully compositional semantics was noted for future work.

Compositionality is important. By it we mean the property that the semantics of a composite construct can determined from the semantics of its parts, so for example, the meaning of the construct $C_1 \mathbin{;;} C_2$ would be determined by the meanings of C_1 and C_2, combined with the meaning of ;;. This property is desirable as without it both the semantics and any reasoning principle based on it would not scale up to large programs or systems.

The structure of the rest of this paper is as follows: we describe some related work (Sect. 2), followed by an introduction to the UTP methodology (Sect. 3). We then explain various aspects of our UTP semantics, touching on labels (Sect. 4), observations (Sect. 5), atomic actions (Sect. 6), and healthiness conditions (Sect. 7). We can then present the semantics in Sect. 8. Finally we discuss some calculations that contribute to the validation of the semantics (Sect. 9) and conclude in Sect. 10.

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 presented in the UTPP paper [23]. This combined guarded commands [9] 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 [10], the "UTP Views" paper by van Staden [21], 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 [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.

All the compositional semantic frameworks we have discussed in this section are based on this notion of sets of transition traces, but we are seeking a semantics based on direct relations between before- and after-program states, without any explicit notion of traces. The reason for this is that the resulting UTP theory will have a form that will make it easier to link to concurrency approaches such as rely-guarantee, or separation logic, that are used with languages that are imperative and program-variable based.

There is however a semantics for shared-variable concurrency that is much closer in form to the one developed in this paper. This is the "actions with axioms" approach of Lamport [18]. In this, the semantics of each language construct is given by a set of axioms, that are 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.

We were not aware of this work when we developed the UTCP theory in this paper, but there are very strong parallels between the features of our semantics and those in [18]. In some sense our semantics is a re-working of his within UTP. We shall point out specific correspondences as we proceed with our presentation.

3 UTP

The Unifying Theories of Programming framework [13] 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:

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

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 $\neg ok \land ok'$ 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 $ok \land P \implies ok' \land Q$, 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 $\mathbf{H}(P) \mathrel{\widehat{=}} ok \implies P$. 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 (Sect. 7) 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 concurrency in UTP [13, 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 necessitates 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 Sects. 5 and 9.

In order to present our UTP semantics of shared-variable concurrency, we have to address an issue that Lamport's semantics [18] faced, namely how to refer to sub-components and their semantics from within a composite. In particular he enunciates a number of principles at the start of his semantics. One identifies the need to know "who" carries out a specific action, while another says that we need to be able to transform a statement about command C into one about command C *within the context of some enclosing construct.*

In the next section we introduce labels and their generators, which are our approach to addressing these concerns. We then follow-up with a description of the observation variables for our theory, how we handle atomic actions, healthiness conditions, and then the semantic definitions.

4 Labels

In order to manage flow-of-control, we need to be able to identify when every construct starts, is running, and ends. In some approaches in the literature, the program syntax allows for and requires explicit labels which are used for this purpose. In our semantics, and that in [19], these identifying labels are generated in a systematic way from the abstract syntax tree. We adopt the idea from [23] that flow of control is managed by an auxiliary variable whose value is the set of all labels of constructs that are able to execute.

We adopted the idea of a label-generator. Given some notion of labels ($l \in Lbs$), we want a notion of a generator ($g \in Gen$) that supports two operations: $new : Gen \rightarrow Lbl \times Gen$ that produces a new label and a new generator; while $split : Gen \rightarrow Gen \times Gen$ 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:

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

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[1]

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

[1] "Split g, take the first one, generate a label and take the resulting generator, split it and take the second, take two new labels and give me the last one".

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 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. In fact this ability to "relocate" is how we manage Lamport's principle that we must be able to talk about command C in the context of an enclosing construct.

5 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 [23] 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".

$$s, s' : \mathit{State} \tag{2}$$

$$\mathit{ls}, \mathit{ls}' : \mathcal{P}\,\mathit{Lbl} \tag{3}$$

Here s and s' denote the before- and after-values of the shared (variable) state, while ls and ls' denote the before- and after-values of the active label-set used for control-flow. In Lamport's semantics [18] a series of temporal logic axioms are provided to track the dynamics of which constructs are starting, in progress, or finishing. We achieve the same effect using the label-sets.

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.

$$\mathit{in}, \mathit{out} : \mathit{Lbl} \tag{4}$$

$$g : \mathit{Gen} \tag{5}$$

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 Sect. 8.3.

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 responsibility of the semantics of language composites.

To summarise, our semantics is will be built using observable variables $s, s', ls, ls', g, in, out$ to describe basic atomic state-change actions that modify global shared state s. The concurrent flow of control will be managed using the 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" [11,12], 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 $x' = x + y \wedge y' = y$[2] . in particular, the notation for sequential composition, $P; Q$, is a shorthand for $\exists obs_m \bullet P[obs_m/obs'] \wedge Q[obs_m/obs]$, 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:

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

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 *semantic* skip ($I\!I$), the unit for semantic sequential composition, as

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

6 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'. Flow of control is managed by keeping a dynamic record of which labels are considered

[2] Assuming `x` and `y` are the only variables.

current, or "enabled". The behaviour of an atomic action is that it exhibits none until its label is enabled. Noting that many atomic actions can be enabled at once, what happens is that one of actions is selected non-deterministically to run. The action so selected transforms the global state, and then the control-flow management marks its label as disabled, and enables labels of atomic action that can immediately follow it according to the control-flow structure of the program.

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:

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

In short: the action when its in-label is in ls, is 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 [23], 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. The result of initialising and running this loop once will be ~~once~~ possible complete execution sequence of the program (assuming it terminates). 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 Eq. 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):

$$A(E \mid a \mid N) \mathrel{\widehat{=}} E \subseteq ls \land a \land ls' = (ls \setminus E) \cup N \quad \langle\!\langle \textsf{A-def} \rangle\!\rangle$$

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 Sect. 9), we need to generalise the basic action idea to an eXtended basic action, where we explicitly identify the labels that we remove (R):

$$X(E \mid a \mid R \mid A) \mathrel{\widehat{=}} E \subseteq ls \land a \land ls' = (ls \setminus R) \cup A \quad \langle\!\langle \textsf{X-def} \rangle\!\rangle$$

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

$$\begin{aligned}
& X(E_1 \mid a \mid R_1 \mid A_1); X(E_2 \mid b \mid R_2 \mid A_2) && \text{《·X-then-X·》} \\
& = E_2 \cap (R_1 \setminus A_1) = \emptyset \\
& \quad \wedge X(E_1 \cup (E_2 \setminus A_1) \ \mid \ a \,;\, b \ \mid \ R_1 \cup R_2 \ \mid \ (A_1 \setminus R_2) \cup A_2)
\end{aligned}$$

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 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. 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 in-between, as an interference step.

7 Healthiness

7.1 Wheels-within-Wheels

We are building a semantics based on predicates that define before-after relations on program state s, s' and label-sets ls, ls', using the static observations to put things in their syntactical context. In order to be able to extract the correct behaviour from this semantics, it was necessary to have a healthiness condition that effectively said that every program component, atomic or composite, has to be viewed as being willing to run as many times as necessary whenever its labels would appear in ls. At its simplest, the semantics required every construct to be embedded in its own infinite loop, to ensure it was always ready to "go". This lead to our use of the phrase "Wheels within Wheels" (WwW) to refer to this principle. This did not mean that everything ran forever, but that, in some sense, it should always be ready.

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.

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

Here we have introduced a *stuttering* step, denoted by UTP's *semantic* skip (II). We note also, that **WwW** is monotonic and idempotent.

It should be noticed that this theory underwent a large number of iterations before the WwW principle was finally elucidated properly and shown to give the right results. The number and complexity of the test calculations needed to debug, develop and validate the theory presented in this paper necessitated the development of a bespoke “UTP Calculator” [6].

7.2 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 separator between things meant to be disjoint, and commas to list subsets and/or set- elements that should be unioned together. So the fragment $A, b \mid M, N \mid x, Y$ 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,

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

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:

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

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.[3]

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

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

[3] 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.

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 or ls' at any point in time:

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

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

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 LE.

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

We note that many of the axioms for a given construct in the semantics of Lamport [18] exist to ensure the same properties regarding construct activation as the healthiness conditions described here.

8 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.

8.1 Atomic Commands

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

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

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 $out \in ls'$.

8.2 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 to do with the fact that, by design, semantic sequential composition ignores the static variables. An expression or predicate 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, and LE satisfies something similar:

$$
\begin{aligned}
K \mathbin{;} K &= K \\
(K \land P) \mathbin{;} Q &= K \land (P \mathbin{;} Q) \;=\; P \mathbin{;} (K \land Q) \\
K \land \mathbf{WwW}(P) &= \mathbf{WwW}(K \land P) \\
(LE \land P) \mathbin{;} (LE \land Q) &= LE \land ((LE \land P) \mathbin{;} (LE \land Q))
\end{aligned}
$$

A substitution is also deemed "ground", if all 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, both disjoint label-set notations, and $\mathbf{WwW}$.

$$
\begin{aligned}
(P \mathbin{;} Q)\gamma &= P\gamma \mathbin{;} Q\gamma && \langle\!\langle\text{seq-gnd-distr}\rangle\!\rangle \\
I\!I\gamma &= I\!I && \langle\!\langle\text{skip-gamma}\rangle\!\rangle \\
\{L_1 \mid \ldots \mid L_n\}\gamma &= \{L_1\gamma \mid \ldots \mid L_n\gamma\} && \langle\!\langle\text{DL-gamma-subst}\rangle\!\rangle \\
[L_1 \mid \ldots \mid L_n]\gamma &= [L_1\gamma \mid \ldots \mid L_n\gamma] && \langle\!\langle\text{LE-gamma-subst}\rangle\!\rangle \\
(\mathbf{WwW}(P))\gamma &= \mathbf{WwW}(P\gamma) && \langle\!\langle\text{WwW-gamma-subst}\rangle\!\rangle
\end{aligned}
$$

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

8.3 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 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. 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 themselves 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. 2.

We will explain the semantics of parallel in more detail, aided by the diagram in Fig. 3. 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 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.

The similarity between our labels and the sub-statement notation of Lamport is quite striking. His parallel construct, called **cobegin**, labels the

$$
\begin{array}{rll}
P \mathbin{;;} Q \mathrel{\widehat=} & \mathbf{W}(P[g_{:1}, \ell_g/g, \mathit{out}] \lor Q[g_{:2}, \ell_g/g, \mathit{in}]) & \langle\!\langle\cdot\mathsf{sem{:}seq}\cdot\rangle\!\rangle \\
P \parallel Q \mathrel{\widehat=} & \mathbf{W}(\ A(\mathit{in} \mid \mathit{ii} \mid \ell_{g1}, \ell_{g2}) \lor & \langle\!\langle\cdot\mathsf{sem{:}par}\cdot\rangle\!\rangle \\
 & \qquad P[g_{1::}, \ell_{g1}, \ell_{g1:}/g, \mathit{in}, \mathit{out}] \lor & \\
 & \qquad Q[g_{2::}, \ell_{g2}, \ell_{g2:}/g, \mathit{in}, \mathit{out}] \lor & \\
 & \qquad A(\ell_{g1:}, \ell_{g2:} \mid \mathit{ii} \mid \mathit{out})\ ) & \\
P + Q \mathrel{\widehat=} & \mathbf{W}(\ P[g_1/g] \lor Q[g_2/g]\ ) & \langle\!\langle\cdot\mathsf{sem{:}NDC}\cdot\rangle\!\rangle \\
P^* \mathrel{\widehat=} & \mathbf{W}(\ A(\mathit{in} \mid \mathit{ii} \mid \ell_g) \lor & \langle\!\langle\cdot\mathsf{sem{:}star}\cdot\rangle\!\rangle \\
 & \qquad A(\ell_g \mid \mathit{ii} \mid \ell_{g:}) \lor & \\
 & \qquad A(\ell_g \mid \mathit{ii} \mid \mathit{out}) \lor & \\
 & \qquad P[g_{::}, \ell_{g:}, \ell_g/g, \mathit{in}, \mathit{out}]\ ) &
\end{array}
$$

Fig. 2. Composite Semantics

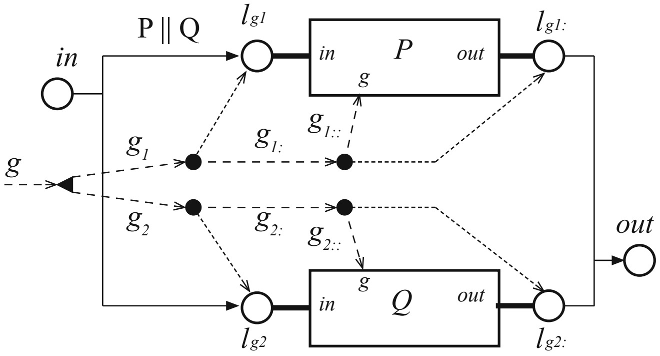

Fig. 3. Label and Generator "plumbing" for $P \parallel Q$.

subcomponents with numbers from 1 upwards. So he refers to P within **cobegin** $P \ \square\ Q$ **coend** as (**cobegin** $P \ \square\ Q$ **coend**$, 1)$. We call it $P[g_{1::}, \ldots/g, \ldots]$. When for a construct P, we assert that $\mathit{in} \in \mathit{ls}$, he uses a predicate $\mathit{at}(P)$. In the parallel case here, an assertion by us that $\ell_{g1} \in \mathit{ls}$, corresponds to his assertion $\mathit{at}($**cobegin** $P \ \square\ Q$ **coend**$, 1)$.

Given that the invariant LE, which is $[\mathit{in} \mid \mathit{labs}(g) \mid \mathit{out}]$, is part of the definition of $\mathbf{W}$, then we have it satisfied, 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 $[\mathit{in} \mid g \mid \mathit{out}]$, but can also infer $[\mathit{in} \mid g_{:1} \mid \ell_g]$ and $[\ell_g \mid g_{:2} \mid \mathit{out}]$.

In summary, we 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 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*.

9 Calculations

Part of the validation of this ~~his~~ semantic theory was by a series of test calculations done to ensure that it was making the right predictions 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.

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

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:

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

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

$$\langle a \rangle = DL \land LE \land (I\!I \lor A(in \mid a \mid out)) \tag{9}$$

Ignoring the healthiness conditions, this boils down to two possible observations we can make of $\langle a \rangle$: either we observe stuttering—no change in state or label-sets ($I\!I$) 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 ($I\!I \lor$ *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

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

Fig. 4. Some Test Calculation Results. Here ab (ba) is short for $a; b$ ($b; a$), and we have omitted the DL and LE invariants for clarity.

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 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. 4. If we look at the result for $\langle a\rangle \mathbin{;;} \langle b\rangle$ we have II, the stuttering step, and $A(\mathit{in} \mid ab \mid \mathit{out})$ which is the complete exection of both actions without interference (mumbling), and $A(\mathit{in} \mid a \mid \ell_g)$ 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 $A(\ell_g \mid b \mid \mathit{out})$, 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 $A(\mathit{in} \mid \mathit{ii} \mid \ell_{g1})$ where the decision to execute a is made. This will remove in from ls if it runs, so disabling the other choice, denoted by $A(\mathit{in} \mid \mathit{ii} \mid \ell_{g2})$. By contrast, in $\langle a\rangle \parallel \langle b\rangle$ the initially enabled action is $A(\mathit{in} \mid \mathit{ii} \mid \ell_{g1}, \ell_{g2})$, which activates both a and b. The control flow action $A(\ell_{g1:}, \ell_{g2:} \mid \mathit{ii} \mid \mathit{out})$ 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 $a \lor b$ and $\langle a \rangle \parallel \langle b \rangle$ to $ab \lor ba$. While this looks cleaner, it has lost too much information, and we cannot compose these further to get correct answers. With the explicitly 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. 4.

10 Conclusions and Future Work

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. As already explained, our semantics has a lot of similarities to the axiomatic action semantics of Lamport [18].

The usefulness of this theory is that it is in a form that makes it very easy for us to specialise it to cover other approaches to concurrency in the Views paper [10], as that paper shows how thay all link to the simple concurrent language whose semantics we have just supplied. These concurrency approaches include Rely-Guarantee [15,22], Owicki-Gries [20], and Concurrent Separation Logic [4]. 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".

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 [14] can be derived from our semantics. We also need to show how the standard operational semantics can be recovered.

We also hope to use this semantics as a baseline for a program to apply UTP to model the various linked approaches discussed in the Views paper [10]. Of particular interest is to explore the connection between UTCP and rely-guarantee [16] approaches. In particular, given our idea of "before" and "after" being able to refer intermediate execution points, and that we can explicitly 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.

Given the similarities between our approach and that of Lamport, it also raises the possibility of bringing our observations and notation closer in line with his.

References

1. Atkinson, D.C., Weeks, D.C., Noll, J.: Tool support for iterative software process modeling. Inform. Softw. Technol. **49**(5), 493–514 (2007). https://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, Montreal, Quebec, Canada, pp. 131–142, 17–19 August 1983
3. 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 1991. LNCS, vol. 527, pp. 111–126. Springer, Heidelberg (1991). https://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
5. 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
6. Butterfield, A.: UTPCalc — a calculator for UTP predicates. In: Bowen, J.P., Zhu, H. (eds.) UTP 2016. LNCS, vol. 10134, pp. 197–216. Springer, Cham (2017). 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, July 2016
8. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic, pp. 366–378. IEEE Computer Society (2007). http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4276538
9. Dijkstra, E.W.: A Discipline of Programming. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs (1976)
10. 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 2013, Rome, Italy, 23–25 January 2013, pp. 287–300. ACM (2013)
11. Hehner, E.C.R.: Predicative programming part i & ii. Commun. ACM **27**(2), 134–151 (1984)
12. Hoare, C.A.R.: Programs are predicates. In: Proceedings 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 (1985)
13. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall International, Englewood Cliffs (1998)
14. Hoare, C.A.R.T., Möller, B., Struth, G., Wehrman, I.: Concurrent kleene algebra. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 399–414. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04081-8_27
15. Jones, C.B.: Developing methods for computer programs including a notion of interference. Ph.D. thesis, University of Oxford, UK (1981)
16. Jones, C.B.: Development methods for computer programs including a notion of interference (PRG-25), p. 265, June 1981
17. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983). http://doi.acm.org/10.1145/69575.69577
18. Lamport, L.: An axiomatic semantics of concurrent programming languages. In: Apt, K.R. (eds.) Logics and Models of Concurrent Systems. NATO ASI Series (Series F: Computer and Systems Sciences), vol. 13, pp. 77–122. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82453-1_4

19. Lamport, L.: Turing lecture: the computer science of concurrency: the early years. Commun. ACM **58**(6), 71–76 (2015). http://doi.acm.org/10.1145/2771951
20. 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
21. Staden, S.: Constructing the views framework. In: Naumann, D. (ed.) UTP 2014. LNCS, vol. 8963, pp. 62–83. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14806-9_4
22. Staden, S.: On rely-guarantee reasoning. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 30–49. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19797-5_2
23. Woodcock, J., Hughes, A.: Unifying theories of parallel programming. In: George, C., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 24–37. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36103-0_5

Author Queries

Chapter 16

Query Refs.	Details Required	Author's response
AQ1	Per Springer style, both city and country names must be present in the affiliations. Accordingly, we have inserted the city and country name in affiliation. Please check and confirm if the inserted city and country name is correct. If not, please provide us with the correct city and country name.	