The Inner and Outer Algebras of Unified Concurrency*

Andrew Butterfield[0000−0002−2337−2101]

Trinity College Dublin, Ireland,
butrfeld@tcd.ie
http://www.scss.tcd.ie/

Abstract. Algebras have always played a critical role in Unifying Theories of Programming, especially in their role in providing the "laws" of programming. The algebraic laws form a triad with two other forms, namely operational and denotational semantics. In this paper we demonstrate that algebras are not just for providing external laws for reasoning about programs. In addition, they can be very beneficial for assisting in the development of theoretical models, most notably denotational semantics. We refer to the algebras used to develop a denotational model as "inner algebras", while the resulting algebraic semantics we consider to be an "outer algebra". In this paper we present a number of inner algebras that arose in the development of a fully compositional denotational semantics, called UTCP, for shared-state concurrency. We explore how these algebras helped to develop (and debug!) the theory, and discuss how they may assist in the ultimate aim of exposing the outer algebra of UTCP, which we expect to be very similar to Concurrent Kleene Algebra.

Keywords: Unifying Theories of Programming · Inner Algebras · Outer Algebras · Shared-Variable Concurrency · Concurrent Kleene Algebras

1 Introduction

The work reported here has been inspired by the "Views" paper [9], which describes how a range of approaches to reasoning about shared-variable concurrency can be mapped down onto instantiations of commutative semi-groups and monoids. The paper introduced a simple language of syntactic commands, and used it as a baseline to connect a wide variety of formal approaches to concurrency. Approaches covered in [9] include various Separation logics [8], type-theories, Owicki-Gries [20], and Rely-Guarantee [17], among others. Our intention in developing a UTP semantics of this 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.

* This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)

Work we did developing a denotational semantics called "Unifying Theory of Concurrent Programs" (UTCP), for the Views command language, using near-homogeneous relations [6], exposed the need for well-defined semantic building blocks, with very well defined properties. Validating the theory as it was developed required a lot of test calculations, to uncover its final form, to such an extent that a rapid "prototyping" calculator was developed to assist in this endeavour [5]. This calculator depended crucially on having well-defined laws and algebras for the semantic building blocks. We use the adjective "inner" to refer to these, and the term "outer" applies to the top-level laws and algebras of the language that is under study.

Most investigations into the relationship between algebraic, and denotational or operational semantics for a language focus on how they relate at the top-level (e.g., [26,25]). In this paper, we focus mainly on those small inner algebras that are very helpful in producing a denotational semantics, against which an (outer) algebraic semantics may be compared.

We next present background material in Section 2 on the Views command language, and other approaches to the semantics of shared-state concurrency. In Section 3 we give a high level overview of the architecture of UTCP, explaining why we have the inner algebras that we do. In Section 4 we explore the inner algebra and laws of UTCP, which we then follow up on in Section 5 with discussion of the state of the outer theory. In Section 6 we look at related work, with an emphasis on those in which there is clear evidence of these inner algebras or laws. Finally, we conclude (Section 7).

2 Background

2.1 View Command Language

The baseline command language from the Views paper assumes an abstract notion of shared state s, and a notion of atomic actions a that non-deterministically modify s. The language syntax then takes atomic commands augmented with `skip` as a building block and introduces operators for sequencing ($;;$), choice ($+$), parallel composition ($\|$) and iteration (*), where all choices are non-deterministic [9]:

$$C ::= \langle a \rangle \mid \texttt{skip} \mid C \;;; C \mid C + C \mid C \parallel C \mid C^*$$

An operational semantics is then defined based on the notion of interleaving of atomic actions. Our notation differs slightly from that in [9] in that we write "$\langle a \rangle$" and ";;" instead of "a" and ";" respectively, for reasons explained in Section 3.

2.2 Denotational Semantics

The UTP theory of concurrent programs (UTCP), whose algebras we discuss here, gives a denotational semantics to the command language above [6]. Denotational semantics of shared variable concurrency are not new, with notable

work in this area having been done by de Boer [1] and Brookes [2]. This resulted in semantics based on the notion of transition traces (TT), which are sequences of state-pairs. A state pair (s_a, s_b) denotes the occurrence of some atomic action that transformed state s_a into state s_b. A transition trace is a sequence of such pairs, with no requirement for the second state of one pair to match the first state of the next. So, if s_i for $i \in 1 \ldots 4$ denotes four different states, then $\langle (s_1, s_2), (s_3, s_4) \rangle$ is valid. It states that the command to which it refers first altered s_1 to s_2, and then something else in the environment ran, changing the state to s_3 along the way, so that when the command resumed to perform its second atomic action, it saw s_3, which it duly converted to s_4. This is basically how the interference of the environment is modelled. The denotational semantics of a command is a set of such traces, with three important healthiness conditions that describe closures:

- *Stuttering:* for any trace $\langle \ldots, (s_1, s_2), (s_3, s_4), \ldots \rangle$ there is also a trace $\langle \ldots, (s_1, s_2), (s, s)(s_3, s_4), \ldots \rangle$ for arbitrary s.
- *Mumbling:* for any trace $\langle \ldots, (s_1, s_2), (s_3, s_4), \ldots \rangle$ where $s_2 = s_3$ then there is also a trace $\langle \ldots, (s_1, s_4), \ldots \rangle$.
- *Interference:* for any trace $\langle \ldots, (s_1, s_2), (s_3, s_4), \ldots \rangle$ there is also a trace $\langle \ldots, (s_1, s_2), (s_5, s_6)(s_3, s_4), \ldots \rangle$ for arbitrary s_5 and s_6.

So the semantics is a set of transition traces closed by adding every possible stuttering action, all possible mumblings, and all possible interference by any possible environment. With the exception of the semantics of a single atomic action, these are all infinitely large sets of traces. These closed sets are fine, when their purpose is to prove that the desired algebraic laws hold for the language under consideration. There is a UTP treatment of Views by van Staden [23] in which he makes use of finite transition traces within an operational calculus.

Another interesting approach to a denotational semantics for shared state concurrency was that reported by Lamport [18]. It is based on the use of temporal logic along with five key ideas, some his, some from others:

1. Being able to identify "who" performs an action
2. Statement assertions true only if true of every program containing that statement.
3. Being able to transform an assertion about a statement into one about a larger statement that contains it.
4. Defining relations between control points as aliasing relations among variables.
5. Allowing *stuttering* actions, to facilitate decomposing atomic actions.

The UTCP theory we describe here was developed before Lamport's work was discovered, but it is interesting to note how our semantics required the rediscovery of concepts analogous to some of the ideas above.

2.3 UTP Action Semantics

A UTP semantics for parallel programming (UTPP) was developed by Woodcock and Hughes [24], that considered a language that required all atomic actions, and

some instances of composite commands to have unique labels. They mapped the language into an action system, where every atomic command became a guarded action, with the guard asserting that the action's label was "enabled", this being modelled by it being present in a global label-set ls. Each guarded action, when enabled, would perform its atomic state-change, remove its enabling label from ls and then add in labels to enable other guarded actions. In effect, the global label set was used to manage flow of control. This theory has the following observations:

$$s, s' : State \tag{1}$$

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

An atomic action, described as a relation $a : State \leftrightarrow State$ with label go, followed by some "after-label" $next$ (say) would exhibit the following behaviour:

$$go \in ls \wedge a \wedge ls' = (ls \setminus \{go\}) \cup \{next\} \tag{3}$$

An action-system is a loop that makes a non-deterministic choice, on each iteration, of one of the currently enabled actions to run. The chosen action will change the state, disable itself, and enable something else.

3 Approach

Our goal for a UTP semantics was to obtain one that was not only compositional (denotational), but was also "local", in the sense that the semantics would only talk about the behaviour of the command under consideration, without being required to also explicitly mention all possible interference. This goal was inspired by the success of separation logic at being able to scale to automatically check very large codebases for pointer errors [21]. A key enabler of that success is that separation logic allows the reasoner to focus on the few pointers actually being manipulated by a program, rather than having to consider (or quantify over) all possible heaps.

We now present a high-level overview of how UTCP is structured, using a simple running example, where a, b and c are arbitrary atomic actions:

$$(\langle a \rangle \;;; \langle b \rangle) \parallel \langle c \rangle \tag{4}$$

The occurrence of action c will be non-deterministically interleaved with those of a and b. Action a will occur before action b. The three possible action sequences (traces) we might observe, assuming no outside interference, are:

$$a; b; c \quad a; c; b \quad c; a; b$$

Here we have represented the sequences using sequential composition (;) which is typically defined in UTP, using O to stand for all observation variables, as:

$$P; Q \mathrel{\widehat{=}} \exists O_m \bullet P[O_m/O'] \wedge Q[O_m/O]$$

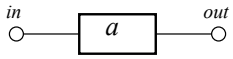

Fig. 1. Atomic action $\langle a \rangle$.

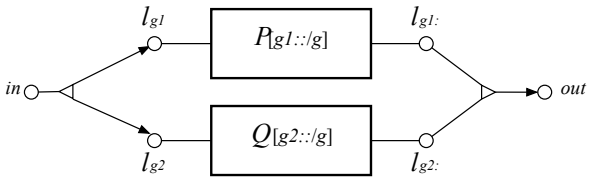

Fig. 2. Parallel composition $P \parallel Q$.

This then raises the question of why we use ;; in our command syntax for sequencing. The simple reason is that ;; is not sequential composition as defined in Eqn. 5, because that definition not only requires the starting state of Q to be the ending state of P, but it also implies that no interference can occur between the end of P and the start of Q. In our example, an execution of $\langle c \rangle$ can come between $\langle a \rangle$ and $\langle b \rangle$. In effect, in any programming language that admits concurrent threads and shared variables, the semantics of sequencing (;;) is *not* sequential composition as defined above. Instead, it is a form of "loose" sequencing in that its first component must terminate before the second can start, but it places no constraints on how the state might be altered in between.

3.1 Labels and Generators

We provide a semantics based on shared-state s and control-flow label-sets ls, in a fashion similar to Eqn. 3. This explains why we use $\langle a \rangle$ to denote the atomic command that performs atomic state-change a, as the latter modifies only s, while the former modifies both s and ls. Unlike UTPP, we do not have explicit labels in our command syntax, but instead we allow the semantic rules to generate appropriate *unique* labels, in a very controlled fashion.

We assume that two label-valued observation variables are associated with every command (atomic or composite), called *in* and *out*. The command starts executing when the label that is the value of *in* is put into the global label set ls. As the command executes, label *in* will be removed from ls, and eventually, as the command terminates, label *out* will appear in ls'. The simplest instance of this is an atomic action, represented symbolically in Fig. 1, with a simplified version of its behaviour shown in Eqn. 5

$$in \in ls \wedge a \wedge ls' = (ls \setminus \{in\}) \cup \{out\} \tag{5}$$

The term above is quite complex, so we introduce a the following shorthand, where E and N are label-sets, and a is an atomic action:

$$A(E|a|N) \mathrel{\widehat{=}} E \subseteq ls \wedge a \wedge ls' = (ls \setminus E) \cup N \tag{6}$$

We typically write E or N by listing the labels, without full set-notation, writing $A(in|a|out)$ rather than $A(\{in\}|a|\{out\})$.

In order to give semantics to composite commands, we need to utilise a way to generate labels that guarantees their uniqueness. We require two ways to use any given generator $G : Gen$: one that generates a label and a new generator ($new : Gen \to Lbl \times Gen$), while the other splits a generator into two ($split : Gen \to Gen \times Gen$). We frequently wish to single out one or other of the components of the result pairs of both the above functions, and so have defined a very compact shorthand notation.

$$\begin{array}{rll} G \in Gen ::= & g & \text{the "root" generator} \\ | & G_{:} & \text{generator after label produced from } G \\ | & G_1 & \text{first generator from split of } G \\ | & G_2 & \text{second generator from split of } G \\ L \in Lbl ::= & \ell_G & \text{label produced by generator } G \end{array}$$

So, for example, $\ell_{g:2}$ is shorthand for:

$$(fst \circ new \circ snd \circ split \circ snd \circ new)(g)$$

Here Gen is an unusual expression language in that it has only *one* variable g, and three postfix operators, $\_:$, $\_1$, and $\_2$. Even more unusual is the label expression language Lbl, which consists solely of the application of prefix function $\ell\_$ to a generator, to get the label it generates. Given a generator G we can (i) get a new label and generator: $G \mapsto (\ell_G, G_{:})$, or (ii) split a generator: $G \mapsto (G_1, G_2)$.

3.2 Semantics with Generators

The way we use generators in our semantics is to introduce a new observation variable g that denotes a label generator that is available. The intuition behind its use is that a composite construct will use g to generate labels for its own use, and split it and pass the resulting generators into its subcomponents. Passing generator G into sub-component P is simply achieved using substitution: $P[G/g]$. In general, a composite may also need to add some *control-flow* actions, which modify ls, but do not alter s. These perform an identity action (ii) on state:

$$ii \mathrel{\widehat{=}} s' = s \tag{7}$$

Let us now consider sequencing $\langle a \rangle$ and $\langle b \rangle$. In effect, we want to arrange it so that the *out* label of $\langle a \rangle$ is the same as the *in* label of $\langle b \rangle$. We do this by taking generator g, and applying the prefix ℓ operator to obtain label ℓ_g, which then is substituted appropriately. We then take the disjunction of the two modified atomic actions, which treats both actions as being part of a non-deterministic choice, as is done in UTPP [24].

$$\langle a \rangle[\ell_g/out] \vee \langle b \rangle[\ell_g/in] \tag{8}$$

$$= A(in|a|out)[\ell_g/out] \vee A(in|b|out)[\ell_g/in] \tag{9}$$

$$= A(in|a|\ell_g) \vee A(\ell_g|b|out) \tag{10}$$

This is not the full picture as there are healthiness conditions to be applied, and one of them is decidedly non-trivial in its effects. These will be presented and discussed later.

Now, consider putting $\langle c \rangle$ in parallel with $\langle a \rangle ;; \langle b \rangle$. We show the general case for arbitrary commands P and Q in Fig. 2. In effect we split a generator, denoted by g into two (g_1, g_2), generate two labels from each to replace the *in* and *out* of P and Q and then add two special control-flow actions. The first replaces the *in* label for the parallel construct as a whole, by the two generated in-labels (l_{g1}, l_{g2}). The second replaces the two generated out-labels $(l_{g1:}, l_{g2:})$ by the *out* label of the whole construct. In essence, given P and Q to be put into parallel, we make use[1] of the following disjunction of two control actions and the two components with appropriate label and generator substitutions:

$$\begin{aligned} & A(in|ii|l_{g1}, l_{g2}) \\ \vee\ & P[g_{1::}, l_{g1}, l_{g1:}/g, in, out] \\ \vee\ & Q[g_{2::}, l_{g2}, l_{g2:}/g, in, out] \\ \vee\ & A(l_{g1:}, l_{g2:}|ii|out) \end{aligned}$$

In our running example, P would be our semantics for $\langle a \rangle ;; \langle b \rangle$ and Q would be $\langle c \rangle$. The latter, after substitution, would appear as $A(l_{g2}|c|l_{g2})$. The former will contain $A(l_{g1}|a|\ell_{g1::})$ and $A(\ell_{g1::}|b|l_{g1:})$, as well as extra components introduced by the healthiness conditions.

3.3 Static and Dynamic Observables

In summary, in addition to observables s, s', ls, ls', we have added in, out, and g. However, this new trio of variables is quite distinct in character, in that they are used within composites to put sub-components into context, by performing substitutions on generators and labels. This contextualisation is *static*, in that it depends on the structure of the program, and it does not change over time. By contrast, observables like s and ls are *dynamic*: they track observables whose values change over time. This distinction is key to making the semantics work.

We now proceed to define sequential composition in our theory in the standard way, provided that we only reference dynamic variables:

$$P; Q \mathrel{\widehat{=}} \exists s_m, ls_m \bullet P[s_m, ls_m/s', ls'] \wedge Q[s_m, ls_m/s, ls] \tag{11}$$

We can also introduce our notion of Skip (II) which is an identity for sequential composition:

$$II \mathrel{\widehat{=}} s' = s \wedge ls' = ls \tag{12}$$

No mention is made in either of the above definitions of *in*, *out*, or g — these are static, and have no dashed counterparts.

We can now present a complete definition of the alphabet of UTCP predicates:

$$s, s' : State \qquad ls, ls' : \mathcal{P} Lbl \qquad in, out : Lbl \qquad g : Gen$$

[1] We have to apply healthiness conditions as well, discussed later.

3.4 Wheels-within-Wheels

Another point to note is that our semantics converts any command into a large disjunction (non-deterministic choice) of atomic and control-flow actions, which precisely correspond to the guarded actions produced in the UTPP semantics [24]. In UTPP, a predicate transformer (run) is applied to the disjunction which initialises the ls and then iterates until/if a distinguished termination label appears in it, generating all possible complete execution traces. This disjunction, produced by both UTPP and UTCP, is static, and in UTPP, run effectively produces the dynamic behaviour. In UTCP, we wanted the predicates generated at every level to capture both static structure, and dynamic behaviour. The intuition was to find a way to "run" at every level in a command program. Each atomic action would be trying to spin continually, awakening when its in label appeared in the label set ls. We need a healthiness condition, to ensure that atomic actions within iterations "stay alive", which basically says the possible behaviour of a command is logically equivalent to making a non-deterministic choice to iterate it zero or more times This healthiness condition has to be applied to the semantics of every command, atomic and composite, leading us to call it "Wheels within Wheels" (**WwW**). Getting the definition of **WwW** right was a major challenge, that drove the development of the rapid-prototype calculator reported in [5]. The key was that we needed to iterate $P \vee II$, which effectively means adding in the possibility of a stuttering step everywhere, leading to the following definition:

$$P^0 \mathrel{\widehat=} II \tag{13}$$

$$P^{i+1} \mathrel{\widehat=} P \mathbin{;} P^i \tag{14}$$

$$\mathbf{WwW}(P) \mathrel{\widehat=} \bigvee_{i \in \mathbb{N}} P^i \tag{15}$$

One key observation here is that with UTCP we need fairly ubiquitous stuttering, just as found in the other compositional theories discussed earlier. Another somewhat striking observation is that we produce a potentially infinite disjunction of P sequentially composed with itself multiple times! This presents quite a challenge for the use of this semantics, and was a key motivation for the UTP Calculator development [5], but it is key to making things work. Given iteration-free programs, the number of iterations actually required is bounded.

Importantly, for a healthiness condition, **WwW** is indeed both idempotent and monotonic:

$$P \sqsubseteq Q \implies \mathbf{WwW}(P) \sqsubseteq \mathbf{WwW}(Q) \tag{16}$$

$$\mathbf{WwW}(\mathbf{WwW}(P)) = \mathbf{WwW}(P) \tag{17}$$

3.5 Label Healthiness

We also need some healthiness conditions on labels, generators, and the global label-set ls. One, "Disjoint labels" (**DL**), requires that $in \neq out$, and neither in

nor *out* appear in $labs(g)$. We introduce more shorthand, using $\{L_1|L_2|\ldots|L_n\}$ for $L_1 \uplus L_2 \uplus \cdots \uplus L_n$, and using G as a shorthand for $labs(g)$ in a context where a label-set is expected rather than a generator. This allows us to write the disjoint label condition as $\{in|g|out\}$. This is a *static* invariant that needs to be satisfied by all healthy P:

$$\mathbf{DL}(P) \mathrel{\widehat{=}} P \wedge \{in|g|out\} \tag{18}$$

Another, "Label Exclusivity" (**LE**), requires that labels *in*, *out* can never occur together in ls, and also never when any member of $labs(g)$ is present. Again, we introduce a shorthand $[L_1|L_2|\ldots|L_n]$ which asserts for any two different L_i and L_j, that $L_i \cap ls \neq \emptyset \implies L_j \cap ls = \emptyset$. We also use $[L_1|L_2|\ldots|L_n]'$ to denote the above assertion with ls replaced by ls' throughout. This is now a *dynamic* invariant that needs to be satisfied by all healthy P:

$$\mathbf{LE}(P) \mathrel{\widehat{=}} P \wedge [in|g|out] \wedge [in|g|out]' \tag{19}$$

Both **DL** and **LE** are clearly idempotent and monotonic w.r.t refinement.

We can now define a top-level healthiness condition called **W**:

$$\mathbf{W}(P) \mathrel{\widehat{=}} \mathbf{DL}(\mathbf{LE}(\mathbf{WwW}(P))) \tag{20}$$

So our full definitions of $\langle a \rangle$ and $P \parallel Q$, and the other constructs, for completeness, can now be shown in Fig. 3.

$$\langle a \rangle \mathrel{\widehat{=}} \mathbf{W}(\ A(in|a|out)\ ) \tag{21}$$

$$\texttt{skip} \mathrel{\widehat{=}} \langle ii \rangle \tag{22}$$

$$P \mathbin{;;} Q \mathrel{\widehat{=}} \mathbf{W}(P[g_{:1}, \ell_g/g, out] \vee Q[g_{:2}, \ell_g/g, in]) \tag{23}$$

$$\begin{aligned} P \parallel Q \mathrel{\widehat{=}} \mathbf{W}(\ & A(in|ii|\ell_{g1}, \ell_{g2}) \vee \\ & P[g_{1::}, \ell_{g1}, \ell_{g1:}/g, in, out] \vee \\ & Q[g_{2::}, \ell_{g2}, \ell_{g2:}/g, in, out] \vee \\ & A(\ell_{g1:}, \ell_{g2:}|ii|out)\ ) \end{aligned} \tag{24}$$

$$\begin{aligned} P + Q \mathrel{\widehat{=}} \mathbf{W}(\ & A(in|ii|\ell_{g1}) \vee A(in|ii|\ell_{g2}) \vee \\ & P[g_{1::}/g] \vee Q[g_{2::}/g] \vee \\ & A(\ell_{g1:}|ii|out) \vee A(\ell_{g2:}|ii|out)\ ) \end{aligned} \tag{25}$$

$$\begin{aligned} P^* \mathrel{\widehat{=}} \mathbf{W}(\ & A(in|ii|\ell_g) \vee \\ & A(\ell_g|ii|\ell_{g:}) \vee \\ & A(\ell_g|ii|out) \vee \\ & P[g_{::}, \ell_{g:}, \ell_g/g, in, out]\ ) \end{aligned} \tag{26}$$

Fig. 3. Command semantics in UTCP

In the following sections, we look at in more detail to uncover the laws and algebras that underpin the semantics just described.

4 Inner Algebras

Here we present some of the algebras and laws that characterise the underlying semantic domains of UTCP.

4.1 Labels and Generators

We need to be sure that however we implement, or model, label generation, that we are sure that unique labels are produced. We can give a minimal specification by positing a function *labs* that takes a generator as input, and returns the set of all labels it could possibly generate, and then requiring that: (i) any label produced from a call to *new* can never occur in the modified generator returned by that call, and (ii) the two generators produced by *split* have disjoint label-sets:

$$labs : Gen \to \mathcal{P}\, Lbl \tag{27}$$

$$\ell_G \notin labs(G_:) \tag{28}$$

$$labs(G_1) \cap labs(G_2) = \emptyset \tag{29}$$

There is a simple way to model/interpret such generators and labels that automatically satisfies the above requirements, plus the following stricter one ($\uplus$ is disjoint union):

$$labs(G) = \{\ell_G\} \uplus labs(G_:) \uplus labs(G_1) \uplus labs(G_2) \tag{30}$$

We simply take labels and generators to be generator expressions themselves, interpreted as strings starting with 'g' and followed by zero or more ':', '1', and '2'. The ℓ operator returns the generator string as the label. The :, 1, and 2 operators append ':', '1', and '2' respectively to the end of the generator string[2]. The advantages of this are two-fold. First, it's simple to describe (and implement,if needed), compared to trying to produce labels as natural numbers (say), that satisfy the requirements above. In particular, there is no need to have a central pool of already generated labels that can be accessed by all the generators that result from *new* and *splits*. Secondly, this interpretation of labels as sequences of symbols that basically record how they were generated from some 'root' generator g, gives us a very easy way to support some of the ideas of Lamport [18], (notably 1,3, and 4, on p3). Given a top-level command P that mentions atomic action $A(\ell_G|a|\ell_{G:})$, then G, as a string, identifies the path from the top-level down to that atomic action, so answering the "who" question (idea 1).

Performing a substitution of G_a for g in G_b ($G_b[G_a/g]$) is equivalent to generator string concatentation, resulting in G_{ab}. Given two generator strings G_p and G_q associated with commands P and Q say, we can answer questions such as: (i) is P a sub-component of Q (G_q a prefix of G_p)? or (ii) do P and Q have a parent component in common, other than the top-level (G_p and G_q have a common prefix)? A key point to understand about the semantics is that the way substitutions for g, *in* and *out* are used maintains this simple relationship between components and sub-components, which makes it easy to facilitate Lamport's idea 3.

[2] A form of Herbrand interpretation!

4.2 Ground Expressions

The distinction between the dynamic observables (s,s',ls,ls') and the static ones (in,out,g) is crucial. In particular, the relationship between substitution and sequential composition is key. Consider applying a substitution σ to the results of a sequential composition of P and Q. On what circumstances should substitution distribute in through such a composition?

$$(P;Q)\sigma =_? P\sigma;Q\sigma$$

If σ involves dynamic observations, then this should clearly not hold. However, all the substitutions in our semantics are used to modify labels in a systematic way down through a sub-component. So if σ only involves static observations, then we do want this distributive property.

Another is issue is our use of the healthiness condition **WwW** in our semantics. This effectively replaces P by a nondeterministic iteration of $P \vee II$, which performs an arbitrary number (zero or more) of sequential compositions. We want certain predicates, and substitutions, to distribute through **WwW**.

To this end, we first define the notion of a *ground term*, as one that only refers to the static observables g, in, and out. A notable example from our semantics is the invariant $\{in|g|out\}$ associated with the **DL** healthiness condition.

Ground predicates K distribute freely through semantic sequential composition:

$$K \wedge (P\ ;\ Q) = (K \wedge P)\ ;\ Q \tag{31}$$
$$= P\ ;\ (K \wedge Q) \tag{32}$$
$$= (K \wedge P)\ ;\ (K \wedge Q) \tag{33}$$

These are all an easy consequence of the way in which we defined sequential composition to only involve the dynamic observables. We also note that sequential composition is idempotent on ground predicates, which clearly indicates that the observables in, out, and g, are truly unchanging.

$$K\ ;\ K = K \tag{34}$$

Finally, we can show that ground predicates K freely distribute in and out of **WwW**:

$$K \wedge \mathbf{WwW}(P) = \mathbf{WwW}(K \wedge P) \tag{35}$$

A *ground substitution* γ is one where the target variables are static, and all the replacement terms (G, I, O) are ground. Such a substitution will have the following most general form:

$$\gamma = [G, I, O/g, in, out] \tag{36}$$

A key property of ground substitutions is that they distribute through sequential composition (and also have no effect on II):

$$(P\ ;\ Q)\gamma = P\gamma\ ;\ Q\gamma \tag{37}$$
$$II\gamma = II \tag{38}$$

They also distribute into the label-set healthiness conditions.

$$\{L_1|\ldots|L_n\}\gamma = \{L_1\gamma|\ldots|L_n\gamma\} \tag{39}$$
$$[L_1|\ldots|L_n]\gamma = [L_1\gamma|\ldots|L_n\gamma] \tag{40}$$

We also have the result that the composition of two ground substitutions is itself ground:

$$[G_1, I_1, O_1/g, in, out]\gamma_2 = [G_1\gamma_2, I_1\gamma_2, O_1\gamma_2/g, in, out] \tag{41}$$

4.3 Sound Substitutions

Consider the ground substitution $[g, \ell_g, \ell_g/g, in, out]$ applied to the label disjointedness assertion $\{in|g|out\}$. We obtain the result $\{\ell_g|g|\ell_g\}$, which violates the **DL** healthiness condition. To prevent this we need the notion of a sound substitution ς, which is a ground substitution where the three replacement expressions themselves collectively satisfy **DL**:

$$\varsigma = [G, I, O/g, in, out] \textbf{ where } \{I|G|O\} \tag{42}$$

We note that soundness is also preserved by substitution composition, and also that every substitution used in the semantics is sound.

The disjoint-label healthiness condition predicate is ground, so **DL** distributes through sequential composition

$$\mathbf{DL}(P\,;Q) = \mathbf{DL}(P)\,;\mathbf{DL}(Q) \tag{43}$$

The label exclusivity invariant mentions dynamic observables ls and ls', so we only get a weaker form of distributivity:

$$\mathbf{LE}(P)\,;\mathbf{LE}(Q) = \mathbf{LE}(\mathbf{LE}(P)\,;\mathbf{LE}(Q)) \tag{44}$$
$$= \mathbf{LE}(P\,;\mathbf{LE}(Q)) \tag{45}$$

4.4 Actions

The core of the UTCP semantics is the notion of a labelled atomic action $A(E|a|N)$. It is enabled when $E \subseteq ls$, and if "chosen" to execute, performs a state change $s \mapsto s'$ that is consistent with the relation a. The semantics of any command reduces to a tree of disjunctions of these, wrapped in the healthiness conditions at every level. The effect of **WwW** is to perform lots of sequential compositions of these with themselves. What is of considerable importance, consequently, is how labelled atomic actions interact with sequential composition.

We start by considering an action composed with itself:

$$A(E|a|N)\,;A(E|a|N) = E \subseteq N \wedge A(E|a^2|N) \tag{46}$$

All atomic actions in the semantics use labelled actions that satisfy **DL**, in which case we have $E \cap N = \emptyset$. For these actions the above self-composition yields **false**. If we consider the full semantics for $\langle a \rangle$:

$$\mathbf{DL}(\mathbf{LE}(\mathbf{WwW}(A(in|a|out))))$$

then the computation of $A(in|a|out)^2$ required by **WwW** is **false**, because we have invariant $\{in|g|out\}$, and so are all the subsequent compositions. So $\mathbf{WwW}(A(in|a|out))$ becomes $II \vee A(in|a|out)$, giving the result:

$$\langle a \rangle = \{in|g|out\} \wedge [in|g|out] \wedge (II \vee A(in|a|out))$$

Unfortunately, our labelled atomic actions are not closed under sequential composition, except under certain conditions, the most notable being when the two actions have no labels in common. In most cases however, we have to introduce an extended form, that differentiates between the enabling labels (E) and those then removed (R). Our basic labelled action is then defined setting $R = E$.

$$X(E|a|R|A) \mathrel{\widehat{=}} E \subseteq ls \wedge a \wedge ls' = (ls \setminus R) \cup A \tag{47}$$

$$A(E|a|N) = X(E|a|E|N) \tag{48}$$

We can calculate that the composition of two extended actions is an extended action, provided the label-sets involved satisfy certain conditions that basically ensure that the second action is enabled after the first one runs.

$$\begin{aligned} &X(E_1|a|R_1|A_1); X(E_2|b|R_2|A_2) \\ &= E_2 \cap (R_1 \setminus A_1) = \emptyset \\ &\quad \wedge X(E_1 \cup (E_2 \setminus A_1) \mid a ;_s b \mid R_1 \cup R_2 \mid (A_1 \setminus R_2) \cup A_2) \end{aligned} \tag{49}$$

Here we introduce sequential composition restricted to s and s':

$$a ;_s b \mathrel{\widehat{=}} \exists s_m \bullet a[s_m/s'] \wedge b[s_m/s], \tag{50}$$

$$ii ;_s a = a = a ;_s ii \tag{51}$$

The result for composing two original atomic actions to produce an extended one is then easily obtained.

$$\begin{aligned} &A(E_1|a|N_1) \; ; \; A(E_2|b|N_2) \\ &= E_2 \cap (E_1 \setminus N_1) = \emptyset \\ &\quad \wedge X(E_1 \cup (E_2 \setminus N_1) \mid a ;_s b \mid E_1 \cup E_2 \mid (N_1 \setminus E_2) \cup N_2) \end{aligned} \tag{52}$$

We finish actions by noting that ground substitutions distribute into their labels:

$$\begin{aligned} A(E|a|N)\gamma &= A(E\gamma|a|N\gamma) \qquad \text{«A-gamma-subs»} \\ X(E|a|R|A)\gamma &= X(E\gamma|a|R\gamma|A\gamma) \end{aligned}$$

4.5 Invariants

Healthiness conditions **DL** and **LE** introduce invariants such as $\{in|g|out\}$, $[in|g|out]$ and/or $[in|g|out]'$. The execution of an atomic action cannot alter the truth of the first one, but it can effect the other two. We require atomic action commands, including control-flow actions, to preserve the **LE** invariant. For basic atomic actions, this is straightforward, and we can show it holds under any sound substitution ς, which covers all the uses of atomic actions as sub-components of composite commands.

$$\{in|g|out\}\varsigma \wedge [in|g|out]\varsigma \wedge A(in|a|out)\varsigma \implies [in|g|out]'\varsigma \tag{53}$$

Finally, given extended actions X_1 and X_2, we have a law about how they interact with law invariants (I_1 and I_2):

$$(I_1 \wedge X_1)\ ;\ (I_2 \wedge X_2) = I_1 \wedge I_2[(ls \backslash R_1) \cup A_1/ls] \wedge (X_1\ ;\ X_2) \tag{54}$$

$$= I_{12} \wedge (X_1\ ;\ X_2) \tag{55}$$

$$\textbf{where } I_{ij} = I_i \wedge I_j[(ls \backslash R_i) \cup A_i/ls] \tag{56}$$

Given that invariants are preserved, as a result of careful theory construction, we might ask if they can be dropped to simplify matters, especially **LE** whose distributivity is limited. Unfortunately, we can't omit them, as they are very useful when doing calculations. It turns out that every instance of $X(\dots)$ that is left over, can be converted back into an equivalent instance of $A(\dots)$, because the invariant gives extra information to allow this simplification. A canonical example of this is $X(L_1|a|L_1, L_2|L_3)$ given invariant $[L_1|L_2|\dots]$. The action is enabled if $L_1 \subseteq ls$, removes L_1 and L_2 from ls, and adds in L_3. However the invariant forbids L_2 from being in ls when this action is enabled, so the removal of L_2 is superfluous. So when enabled, the above action is equivalent to $X(L_1|a|L_1|L_3)$, which is the same as $A(L_1|a|L_3)$, by Eqn. 48.

5 Outer Algebras

The notion of Concurrent Kleene Algebra (CKA) $(A, +, *, ;, ^{\circledast}, ^{\odot}, 0, 1)$ is being put forward as a baseline for the semantics of all programming languages[16], and is defined as a bi-Kleene algebra over concurrent monoid $(A, *, ;, 1, \leq)$. In that paper, a following rough correspondence between CKA operators and those of CSP are given: $+$ is non-deterministic choice, ; is sequential composition, $*$ is some form of parallelism, $\leq$ is refinement, 1 is SKIP, 0 is miracle, and the circled operators are iterated parallel and sequential composition.

The command language does not have all of those operators, and so we cannot claim that its semantics forms a CKA. If we define

$$\texttt{skip} \mathrel{\widehat{=}} \langle ii \rangle \tag{57}$$

$$P \sqsubseteq Q \mathrel{\widehat{=}} P = P + Q \tag{58}$$

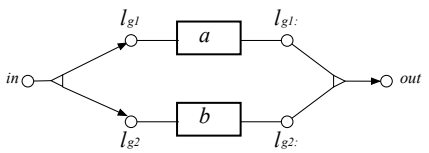

Fig. 4. $\langle a \rangle \parallel \langle b \rangle$

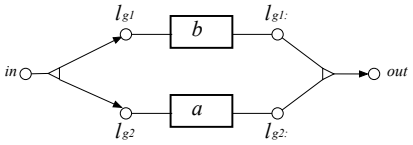

Fig. 5. $\langle b \rangle \parallel \langle a \rangle$.

then we can posit the following laws:

$$\begin{aligned}
\texttt{skip} \mathbin{;;} P &= P \\
P \mathbin{;;} \texttt{skip} &= P \\
P \mathbin{;;} (Q \mathbin{;;} R) &= (P \mathbin{;;} Q) \mathbin{;;} R \\
P \parallel Q &= Q \parallel P \\
P \parallel (Q \parallel R) &= (P \parallel Q) \parallel R \\
P + P &= P \\
P + Q &= Q + P \\
P + (Q + R) &= (P + Q) + R \\
P + Q &\sqsubseteq P \\
P \sqsubseteq Q &\equiv (P + Q) = Q \\
P^* &= \texttt{skip} + P \mathbin{;;} P^* \\
P_1 \mathbin{;;} (P_2 \parallel P_3) &\sqsubseteq (P_1 \mathbin{;;} P_2) \parallel P_3 \\
P_1 \mathbin{;;} P_2 &\sqsubseteq P_1 \parallel P_2
\end{aligned}$$

Here we discuss how we might proceed to prove that these laws are a consequence of our semantics. The best approach is to explore simple examples of the above laws using atomic actions. We will consider the follow three in order, chosen to reveal key issues we have to tackle.

$$\langle a \rangle \parallel \langle b \rangle = \langle b \rangle \parallel \langle a \rangle \tag{59}$$

$$\langle a \rangle \mathbin{;;} (\langle b \rangle \mathbin{;;} \langle c \rangle) = (\langle a \rangle \mathbin{;;} \langle b \rangle) \mathbin{;;} \langle c \rangle \tag{60}$$

$$\langle a \rangle + \langle a \rangle = \langle a \rangle \tag{61}$$

The semantics of parallel can be pictured as per Figure 2 on p5. The instantiation of this for both sides of the parallel commutativity law example (Eqn. 59) are shown in Figures 4 and 5. We can see that the only difference is that the ℓ_{gNx} labels have been swapped around. This suggests that we should either ignore the particular labels and just look at the structure, of perhaps assume that any bijective mapping of labels has no effect on behaviour.

When we consider the associativity of sequencing (Eqn. 60), the two sides are shown in Figures 6. and 7. Again there is an obvious one-to-one mapping from labels that makes them equivalent.

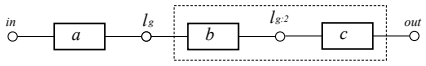

Fig. 6. $\langle a \rangle \;;;\; (\langle b \rangle \;;;\; \langle c \rangle)$.

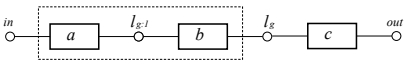

Fig. 7. $(\langle a \rangle \;;;\; \langle b \rangle) \;;;\; \langle c \rangle$.

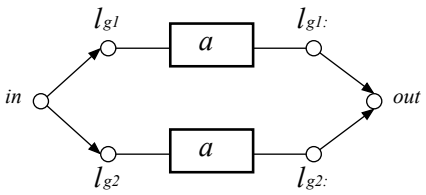

Fig. 8. $\langle a \rangle + \langle a \rangle$.

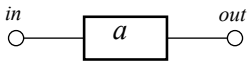

Fig. 9. $\langle a \rangle$.

Things get more complicated when we compare the diagrams (Figures 8 and 9) for the third example, the idempotence of choice (Eqn. 61). This requires more thought: only one of the atomic actions in the lefthand side will run. Unlike the parallel example, where the "production" of both ℓ_{g1} and ℓ_{g2} result from the consumption of in, here we have have two distinct "edges" from label in, so once in is in ls, both edges are enabled, but only one is chosen non-deterministically, so either ℓ_{g1} or ℓ_{g2} are enabled, but not both. As in will be removed from ls, so there is no immediate chance of the other option being enabled. What we have to realise is that control-flow actions are important but from an external observer's perspective, as long as they are well-behaved, the precise details do not matter. In effect this means that our notion of similarity of these graphs needs to be based on a form of bijection-like relation between non-empty sets of labels, where sets being related may not have the same size. The bijection-like aspect would arise in that if L_1 and L_2 are related, then none of the elements of either set may occur in any other pair of related sets. In this example we would propose the relation:

$$\{(\{in, \ell_{g1}, \ell_{g2}\}, \{in\}), (\{\ell_{g1:}, \ell_{g2:}, out\}, \{out\})\}$$

The upshot of all of this, is that on possible plan to prove the laws has two steps: the first is to show that the laws always induce pairs of graphs like the above related by some relational bijection. The second is to show that these graphs induce the behaviour that results from calculating with the semantics.

There is another alternative to be considered: the UTCP semantics define predicates, but their structure is very graph-like. We can view the labelled nodes as vertices of the graph, and labelled actions as action-labelled edges connecting a set of "input" vertices to a set of "output" vertices. We can imagine a vertex coloured black or white to indicate if its label is present in ls. With this graph interpretation of the denotational semantics (predicate) rules, we should be able

to produce an operational semantics. This may provide an alternative route to proving the laws.

6 Related Work

The most obvious use of inner algebras in UTP can be found in those theories, usually of concurrency, that make use of trace observations. We can find these in the UTP book, for example the definition of the trace merge operator [15, Defn 8.19, p203]. We also see a variant of it used in *Circus*[19]. It is particulary notable in any work adding time to traces, such as *Circus*-time [22], "slotted"-*Circus*[7,13,14], and recent work on trace algebras being used for hybrid semantics [10]. A particularly interesting example of inner algebras and laws, comes from work mechanising UTP in Isabelle/HOL [12]. In this, the traditional pre/post divide becomes a pre/peri/post divide, in which the behaviours when waiting for an external event are captured as distinct peri-conditions [11]. Certain idioms commonly used when defining pre-, peri-, and post-conditions are abstracted out and shown to obey useful laws. These prove to be very valuable for high speed automated proof.

7 Conclusions

While describing work to develop and validate a denotational semantics for the command language in UTP, we have discovered the value of algebra as a tool to help develop such theories. This has been driven by the need to manage the complexity inherent in the underlying semantic domains. It is particularly helpful when the area has conceptual difficulties, and you need healthiness conditions, such as **WwW**, that are highly counter-intuitive. The need for some automation to help assess emerging theories drove the development of the "UTP calculator" described in [5]. What became very clear from that work is that good algebra design leads to very effective and fast calculation, with a lot of scope for automation. The calculator can have been considered a stop-gap measure, but has inspired a complete re-design of the Saoithín/UTP2 theorem prover, originally described in [3,4]. This new version, being developed at https://github.com/andrewbutterfield/reasonEq will support both proof and calculation, with scope for considerable automation.

References

1. de Boer, F.S., Kok, J.N., Palamidessi, C., Rutten, J.J.M.M.: The failure of failures in a paradigm for asynchronous communication. In: Baeten, J.C.M., Groote, J.F. (eds.) CONCUR '91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991, Proceedings. Lecture Notes in Computer Science, vol. 527, pp. 111–126. Springer (1991). https://doi.org/10.1007/3-540-54430-5˙84, http://dx.doi.org/10.1007/3-540-54430-5_84

2. Brookes, S.D.: Full abstraction for a shared-variable parallel language. Inf. Comput. **127**(2), 145–163 (1996). https://doi.org/10.1006/inco.1996.0056, http://dx.doi.org/10.1006/inco.1996.0056
3. Butterfield, A.: Saoithín: A theorem prover for UTP. In: Unifying Theories of Programming - Third International Symposium, UTP 2010, Shanghai, China, November 15-16, 2010. Proceedings. pp. 137–156 (2010). https://doi.org/10.1007/978-3-642-16690-7˙6, http://dx.doi.org/10.1007/978-3-642-16690-7_6
4. Butterfield, A.: The logic of $U \cdot (TP)^2$. In: Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, France, August 27-28, 2012, Revised Selected Papers. pp. 124–143 (2012). https://doi.org/10.1007/978-3-642-35705-3˙6, http://dx.doi.org/10.1007/978-3-642-35705-3_6
5. Butterfield, A.: UTPCalc - A Calculator for UTP Predicates. In: Bowen, J.P., Zhu, H. (eds.) Unifying Theories of Programming - 6th International Symposium, UTP 2016, Reykjavik, Iceland, June 4-5, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10134, pp. 197–216. Springer (2016). https://doi.org/10.1007/978-3-319-52228-9˙10, https://doi.org/10.1007/978-3-319-52228-9_10
6. Butterfield, A.: UTCP: compositional semantics for shared-variable concurrency. In: da Costa Cavalheiro, S.A., Fiadeiro, J.L. (eds.) Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10623, pp. 253–270. Springer (2017). https://doi.org/10.1007/978-3-319-70848-5_16, https://doi.org/10.1007/978-3-319-70848-5_16
7. Butterfield, A., Sherif, A., Woodcock, J.: Slotted-circus. In: Davies, J., Gibbons, J. (eds.) Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4591, pp. 75–97. Springer (2007). https://doi.org/10.1007/978-3-540-73210-5_5, https://doi.org/10.1007/978-3-540-73210-5_5
8. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: 22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings. pp. 366–378. IEEE Computer Society (2007)
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). https://doi.org/10.1145/2480359.2429104
10. Foster, S., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying theories of time with generalised reactive processes. Inf. Process. Lett. **135**, 47–52 (2018). https://doi.org/10.1016/j.ipl.2018.02.017, https://doi.org/10.1016/j.ipl.2018.02.017
11. Foster, S., Ye, K., Cavalcanti, A., Woodcock, J.: Calculational verification of reactive programs with reactive relations and kleene algebra. In: Desharnais, J., Guttmann, W., Joosten, S. (eds.) Relational and Algebraic Methods in Computer Science - 17th International Conference, RAMiCS 2018, Groningen, The Netherlands, October 29 - November 1, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11194, pp. 205–224. Springer (2018). https://doi.org/10.1007/978-3-030-02149-8_13, https://doi.org/10.1007/978-3-030-02149-8_13
12. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering 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. 21–41. Springer (2014). https://doi.org/10.1007/978-3-319-14806-9˙2, http://dx.doi.org/10.1007/978-3-319-14806-9_2

13. Gancarski, P., Butterfield, A.: The Denotational Semantics of slotted-Circus. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 451–466. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_29
14. Gancarski, P., Butterfield, A.: Prioritized slotted-Circus. In: Cavalcanti, A., Déharbe, D., Gaudel, M., Woodcock, J. (eds.) Theoretical Aspects of Computing - ICTAC 2010, 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6255, pp. 91–105. Springer (2010). https://doi.org/10.1007/978-3-642-14808-8_7
15. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
16. Hoare, C.A.R., Möller, B., Struth, G., Wehrman, I.: Concurrent kleene algebra and its foundations. The Journal of Logic and Algebraic Programming **80**(6), 266 – 296 (2011). https://doi.org/https://doi.org/10.1016/j.jlap.2011.04.005, http://www.sciencedirect.com/science/article/pii/S1567832611000166, relations and Kleene Algebras in Computer Science
17. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983). https://doi.org/10.1145/69575.69577, http://doi.acm.org/10.1145/69575.69577
18. Lamport, L.: An Axiomatic Semantics of Concurrent Programming Languages, pp. 77–122. Springer Berlin Heidelberg, Berlin, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82453-1˙4, https://doi.org/10.1007/978-3-642-82453-1_4
19. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Asp. Comput **21**(1-2), 3–32 (2009), http://dx.doi.org/10.1007/s00165-007-0052-5
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, https://doi.org/10.1007/BF00268134
21. Pym, D., Spring, J.M., O'Hearn, P.: Why separation logic works. Philosophy & Technology (May 2018). https://doi.org/10.1007/s13347-018-0312-8
22. Sherif, A., He, J.: Towards a time model for circus. 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. 613–624. Springer (2002). https://doi.org/10.1007/3-540-36103-0_62, https://doi.org/10.1007/3-540-36103-0_62
23. van Staden, S.: Constructing the views framework. In: Naumann, D. (ed.) Unifying Theories of Programming - 5th International Symposium, UTP 2014, Singapore, May 13, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8963, pp. 62–83. Springer (2014). https://doi.org/10.1007/978-3-319-14806-9˙4, http://dx.doi.org/10.1007/978-3-319-14806-9_4
24. Woodcock, J., Hughes, A.P.: Unifying theories of parallel programming. In: George, C., Miao, H. (eds.) Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2495, pp. 24–37. Springer (2002). https://doi.org/10.1007/3-540-36103-0˙5

25. Zhu, H., He, J., Qin, S., Brooke, P.J.: Denotational semantics and its algebraic derivation for an event-driven system-level language. Formal Asp. Comput **27**(1), 133–166 (2015), http://dx.doi.org/10.1007/s00165-014-0309-8
26. Zhu, H., Yang, F., He, J.: Generating denotational semantics from algebraic semantics for event-driven system-level language. In: Qin, S. (ed.) Unifying Theories of Programming - Third International Symposium, UTP 2010, Shanghai, China, November 15-16, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6445, pp. 286–308. Springer (2010). https://doi.org/10.1007/978-3-642-16690-7˙15, https://doi.org/10.1007/978-3-642-16690-7_15