Communicating Process Architectures – 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

A Predicate Transformer Semantics for a Concurrent Language of Refinement

Ana CAVALCANTI and Jim WOODCOCK
Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, CT2 7NF, England

Abstract. *Circus* is a combination of Z and CSP; its chief distinguishing feature is the inclusion of the ideas of the refinement calculus. Our main objective is the definition of refinement methods for concurrent programs. The original semantic model for *Circus* is Hoare and He's unifying theories of programming. In this paper, we present an equivalent semantics based on predicate transformers. With this new model, we provide a more adequate basis for the formalisation of refinement and verification-condition generation rules. Furthermore, this new framework makes it possible to include logical variables and angelic nondeterminism in *Circus*. The consistency of the relational and predicate transformer models gives us confidence in their accuracy.

1 Introduction

Modern computing systems typically run on distributed, heterogeneous networks, and are subject to complex constraints on functionality, performance, fault tolerance, security, and timing. If we want such systems to be dependable, then we must address all these issues. There has been much progress in providing sound mathematical foundations for these different aspects of complex systems, but it is usually done in relative isolation. This is because researchers apply the principle of separation of concerns, allowing them to find a thorough solution to a particular problem without being distracted by others. Subsequent researchers can then build on this fundamental research by composing its theories.

Two examples of this separation of concerns lie in the theory of program specification and development. First, research on state-based, model-oriented specification languages originally focused on the specification and refinement of sequential software, avoiding the complications of concurrency and distribution. Second, when research on process algebras started in the 1970s, the major schools focused on studying the semantics and theory of concurrency and communication, abstracting from the details of data types and their operations. In both cases, researchers knew that they would have to address the wider concerns in order for the techniques to scale up to describing industrial-sized systems.

Recently, there has been considerable interest in bringing these two strands of research together, and in particular combining Z [1, 2] and CSP [3, 4] in various ways; Fischer has surveyed some of this work [5]. It is clear, however, that little has been accomplished in understanding the formal development of programs starting from specifications in these combined formalisms.

Circus [6, 7, 8] is a language for writing specifications, designs, and programs for concurrent, communicating systems. It combines the languages of Z, CSP, the refinement calculus [9], and guarded commands [10]. As such, it is very convenient for capturing and reasoning about static, dynamic, and reactive aspects of concurrent and distributed systems, such as may be written in occam and Java.

The complete integration of the four languages involved in *Circus* has been achieved by giving a single denotational semantics that describes all their constructs, in the manner of

Hoare and He's unifying theories of programming [11]. In Hoare and He's work, a number of different paradigms for programming are studied as a series of linked theories, each formalised in an alphabetised variant of Tarski's relational calculus.

A long-standing trend in the formal development and verification of programs is the use of predicate transformers, either directly or as a foundation [10, 12, 13, 14]. In the unifying theory, the relational model for sequential programs is linked to a weakest-precondition model. Later, the relational model is augmented to cope with concurrency and communication, and the link to the weakest-precondition setting turns out not to be valid for the augmented model.

We present a new predicate transformer: the *weakest reactive precondition*. It characterises the weakest precondition that guarantees that a given condition holds in all observable states of a reactive program. We define the weakest reactive precondition of a unifying theory relation that defines a reactive system. From this, we calculate a weakest reactive precondition semantics for *Circus*.

This new semantic model is a convenient step towards the complete justification of our extension to an existing refinement calculus for Z [15] that includes all *Circus* constructs [16, 17]. Moreover, the weakest precondition semantics is a natural starting point for the proposal of rules for verification-condition generation for *Circus*. Finally, as an added benefit to the effort of calculating a new semantics for *Circus*, we now have the possibility of modelling logical constants and angelic nondeterminism, which are important for refinement techniques. The two models also provide confirmation for the precision of each other.

In [18], laws are presented that completely characterise occam's semantics, and which are cast in terms of the denotational semantics of [19], although no proof of equivalence was carried out. The laws presented in that work, however, are equalities; they aim at characterising the semantics of the language, instead of supporting the development of programs.

In the next section, we give an overview of *Circus*. Section 3 presents the basic concepts of the unifying theory. The weakest precondition semantics of *Circus* is the subject of Section 4. Finally, in Section 5 we present our conclusions.

2 *Circus*

A *Circus* program is a sequence of paragraphs; each of these may either be a Z paragraph, a channel definition, a channel set definition, or a process definition. In the BNF description of the syntax of *Circus* in Figure 1, we omit part of the syntax of processes (Proc) and the syntax of communications (Comm), which we exemplify later on in this section.

CircusPar* denotes a possibly empty list of elements from the category CircusPar; similarly for PPar*. N^+ denotes a comma-separated list of valid Z identifiers (elements of N), and similarly for Exp^+. The syntactic categories Par, Schema-Exp, Exp, Pred, and Decl include the Z paragraphs, schema expressions, expressions, predicates, and declarations; their definitions can be found in [1]. Finally, the syntactic category CSExp of channel set expressions contains the empty set of channels $\{\!|\ |\!\}$, channel enumerations enclosed in $\{\!|$ and $|\!\}$, and set expressions formed by the usual set operators.

A process encapsulates state and behaviour. The basic form of process definition describes the state and operations, mainly as in a standard Z specification. In the context of *Circus*, the operations are called actions and can be specified using schemas, CSP operators, and guarded commands. The predicate transformer semantics calculated here characterises the behaviour of actions.

By way of illustration, we consider a little example due to Hoare [3]. The process *WSum* below inputs natural numbers from a channel *in*, and outputs through a channel *out* the

Program	::=	CircusPar*
CircusPar	::=	Par | ChanDef | ChanSetDef | ProcDef
ChanDef	::=	**channel** CDecl
CDecl	::=	SimpleCDecl | SimpleCDecl; CDecl
SimpleCDecl	::=	N+ | N+ : Exp | Schema-Exp
ChanSetDef	::=	**chanset** N == CSExp
ProcDef	::=	**process** N ≙ Proc
Proc	::=	**begin** PPar* • Action **end** | Proc □ Proc
	|	...
PPar	::=	Par | N ≙ Action
Action	::=	Schema-Exp | CSPActionExp | Command
CSPActionExp	::=	*Skip* | *Stop* | *Chaos*
	|	Comm → Action | Pred & Action | Action; Action
	|	Action □ Action | Action ⟦ CSExp ⟧ Action | μ N • Action
	|	Decl • Action | Action(Exp+)
Command	::=	N+ : [Pred, Pred] | N+ := Exp+
	|	**if** GuardedActions **fi** | **var** Decl • Action
GuardedActions	::=	Pred → Action | Pred → Action □ GuardedActions

Figure 1: *Circus* syntax

weighted sum of its current and previous input.

$\textbf{channel}\ in, out : \mathbb{N};$

The weights are defined as constants, using the Z notation.

$|\ a, b : \mathbb{N}$

The state S records its previous input *last* and the value to be output *val*.

$\textbf{process}\ WSum \mathrel{\widehat=} \textbf{begin}$

$S \mathrel{\widehat=} [\, last, val : \mathbb{N} \,]$

We present the action *Compute*, which takes an input $x?$ and updates the state accordingly.

$$Compute \mathrel{\widehat=} [\, \Delta S;\ x? : \mathbb{N} \mid val' = a * last + b * x? \land last' = x? \,]$$

There is a nameless action at the end of a process description, which defines its behaviour; we refer to this action as the main action of the process. In our example, it is as follows.

$$\bullet\ (\, \mu X \bullet in?x \rightarrow Compute;\ out!val \rightarrow X \,)$$

end

We use the prefixing, sequence, and recursion constructs of CSP. First the input is taken through the channel *in* and into the input variable x. Afterwards, the schema action previously defined is used to update the state. In sequence, the *val* component of the state is output through *out* and the process recurses. We explain the other action operators below through a more substantial example, and in Section 4, as we present their semantics.

The CSP operators can also be used to combine processes: their states are conjoined and their main actions are combined using the CSP operator applied. The weakest precondition of a process is that of its main action. The following example, inspired by the Sieve of Eratosthenes presented in [20], illustrates the use of processes.

Example: The objective of the Sieve of Eratosthenes is to produce the list of prime numbers. In our example, the list is output through the channel *out*.

$$\textbf{channel}\ out : \mathbb{N}$$

We use the set *primes* that contains all the prime numbers. Its Z specification is as follows.

$$primes == \{\, n : \mathbb{N} \mid (\forall\, m : \mathbb{N} \bullet m\ divides\ n \Rightarrow m = 1 \lor m = n) \}$$

In this definition, we use the function *_divides_*, with the obvious meaning: *m divides n* if and only if *n* is a multiple of *m*. The definition of this function in Z is simple and omitted. So, a number is prime if its only divisors are 1 and itself.

The specification is a parametrised process, *Primes*. Its parameter *l* limits the size of the list of prime numbers: we are interested only in those primes less than or equal to *l*.

$$
\begin{array}{l}
\textbf{process}\ Primes \mathrel{\widehat=} l : \mathbb{N} \bullet \textbf{begin}\\
\quad PrimesState \mathrel{\widehat=} [\, s : \mathbb{F}\,\mathbb{N} \,]\\
\quad PrimesInit \mathrel{\widehat=} [\, PrimesState' \mid s' = (2 \mathinner{..} l) \cap primes \,]\\
\quad Output \mathrel{\widehat=} [\, \Delta PrimesState;\ m! : \mathbb{N} \mid s \neq \emptyset \land m! = \min s \land s' = s \setminus \{\, m! \,\} \,]\\
\quad \bullet\ PrimesInit;\\
\qquad \mu\, X \bullet\ \ \textbf{if}\ s = \emptyset \rightarrow Skip\\
\qquad\qquad\quad [\!] \ s \neq \emptyset \rightarrow \textbf{var}\ m : \mathbb{N} \bullet Output;\ out!m \rightarrow X\\
\qquad\qquad\quad \textbf{fi}\\
\textbf{end}
\end{array}
$$

The process's only state component, specified in *PrimesState*, is a finite set *s* of natural numbers. The main action describes the behaviour of the process: first it initialises the state with *PrimesInit*; then it recursively executes a conditional action. The notation that we use is that of Dijkstra's guarded commands [10].

The initialisation sets *s* to the set of numbers that are both between 2 and *l* and also prime. The recursive action then outputs these numbers in ascending order. If *s* is empty, then the task is finished, and the recursion terminates. Otherwise, a local variable *m* is declared to hold the next output, which is selected by the operation *Output*. The schema *Output* requires that *s* must not be empty, and the after-value of *m*—here denoted by $m!$ to emphasise that it is a result of the operation—is set to the minimum value in *s*; $m!$ is then removed from *s* to form the after-value of the state, s'.

Primes is refined to the process *Eratosthenes*, defined below, which specifies a concurrent implementation of the Sieve of Eratosthenes algorithm. It is defined as the parallel composition of other processes, in the manner of a systolic array.

The first process, which we call *Start*, outputs 2, which is the first prime, through *out*. Afterwards, it outputs the list of odd numbers, or rather, the numbers that are not multiples of 2, through a channel *sievein*.

$$\textbf{channel}\ sievein : \mathbb{N}$$

$$
\begin{array}{l}
\textbf{process}\ Start \mathrel{\widehat=} \textbf{begin}\\
\quad StartState \mathrel{\widehat=} [\, n : \mathbb{N} \,]\\
\quad \bullet\ out!2 \rightarrow n := 3;\ (\mu\, X \bullet sievein!n \rightarrow n := n + 2;\ X)\\
\textbf{end}
\end{array}
$$

The second process is a *Filter*. It takes a prime p through the channel *sievein*, outputs it, and then outputs the list of numbers that are not multiples of p, through a channel *sieveout*.

$$
\begin{array}{l}
\textbf{channel}\ \mathit{sieveout} : \mathbb{N} \\
\textbf{process}\ \mathit{Filter} \mathrel{\widehat=} \textbf{begin}\ \bullet \\
\quad \mathit{sievein}?p \rightarrow \mathit{out}!p \rightarrow \\
\quad \mu X \bullet \mathit{sievein}?m \rightarrow \ \begin{array}[t]{l} \textbf{if}\ p\ \mathit{divides}\ m \rightarrow X \\ [\!]\ \neg\,(p\ \mathit{divides}\ m) \rightarrow \mathit{sieveout}!m \rightarrow X \\ \textbf{fi} \end{array} \\
\textbf{end}
\end{array}
$$

The idea is that *Start* sends through *sievein* the number 3, which is the next prime after 2, and all the odd numbers. *Filter* outputs 3 and filters out from *sievein* the multiples of 3. The multiples of 2 have already been removed by *Start*, so we are left only with numbers that are neither a multiple of 2 nor 3. The list of such numbers are output through *sieveout*.

We use a piping operator to take the list output through *sieveout* as input to *Filter* again. This operator takes the form $[c_1 \leftrightarrow c_2]\ n \bullet P$, and we define it as follows, for $n > 0$, and for a channel m not used in P.

$$
\begin{array}{l}
[c_1 \leftrightarrow c_2]\ n \bullet P \mathrel{\widehat=} \\
\quad \textbf{var}\ v : \mathbb{N} \bullet v := n; \\
\qquad \mu X \bullet\ \begin{array}[t]{l} \textbf{if}\ v = 1 \rightarrow P \\ [\!]\ v > 1 \rightarrow v := v - 1;\ (P[m/c_1] \mathrel{[\![} \{\!|\ m\ |\!\} \mathrel{]\!]} X[m/c_2]) \setminus \{\!|\ m\ |\!\} \\ \textbf{fi} \end{array}
\end{array}
$$

The operator forms n copies of P, and links them together by connecting the c_1 channel of the i-th process to the c_2 channel of the $(i+1)$-th process. The definition achieves this by recursion, limited by the local variable v. On every recursive step, except the last, a copy of P is connected to its right neighbour. The copy has its c_1 channel renamed to m; the neighbour (the recursive call of X) has its c_2 channel also renamed to m. They are then composed in parallel, communicating only through m, which is hidden to prevent interference. The result is a process with a single c_2 channel on the left and a single c_1 channel on the right. Its internal structure is a linear sequence of processes communicating pairwise on private channels. Of course, P may have other channels, and they are not linked in any way by this operator.

For example, *COPY* (taken from [3]) is a one-place buffer that repeatedly copies its inputs from the *left* channel and outputs them on the *right* channel:

$$
\begin{array}{l}
\textbf{channel}\ \mathit{left}, \mathit{right} : \mathbb{N} \\
\textbf{process}\ \mathit{COPY} \mathrel{\widehat=} \textbf{begin}\ \bullet\ (\mu X \bullet \mathit{left}?x \rightarrow \mathit{right}!x \rightarrow X)\ \textbf{end}
\end{array}
$$

We construct a three-place buffer as $[\mathit{right} \leftrightarrow \mathit{left}]\ 3 \bullet \mathit{COPY}$. Once the recursion has been completely unfolded, v is irrelevant, so the process is equivalent to the expression:

$$
\begin{array}{l}
(\ \mathit{COPY}[m/\mathit{right}] \\
\quad [\![\{\!|\ m\ |\!\}]\!] \\
\quad (\ (\ \mathit{COPY}[m/\mathit{right}] \mathrel{[\![} \{\!|\ m\ |\!\} \mathrel{]\!]} \mathit{COPY}[m/\mathit{left}]\ ) \setminus \{\!|\ m\ |\!\}\ )[m/\mathit{left}] \\
\ ) \setminus \{\!|\ m\ |\!\}
\end{array}
$$

This three-place buffer has two external channels, *left* and *right*, and two internal ones. Data arrives on the *left* channel, makes its way via the two internal channels, and finally appears on the *right* channel. The result is a pipeline of replicated components, like a systolic array. In a similar way, we construct our sieve as a pipeline from a series of l filters.

$$
\textbf{process}\ \mathit{Sieve} \mathrel{\widehat=} l : \mathbb{N} \bullet (\ [\mathit{sieveout} \leftrightarrow \mathit{sievein}]\ l \bullet \mathit{Filter}\ )
$$

At the far end of the pipeline we have the process *Finish*, which simply takes a value through *sieveout* and outputs it.

$$\textbf{process}\ Finish \mathrel{\widehat=} \textbf{begin} \bullet sieveout?n \rightarrow out!n \rightarrow Skip\ \textbf{end}$$

Now we can connect together our components to form the implementation.

$$\begin{array}{l}\textbf{process}\ Eratosthenes \mathrel{\widehat=} l : \mathbb{N} \bullet \\ \quad (\, Start \llbracket\, \{\!|sievein|\!\}\, \rrbracket\, Sieve(l) \llbracket\, \{\!|sieveout|\!\}\, \rrbracket\, Finish\,) \setminus \{\!|sievein, sieveout|\!\}\end{array}$$

The *out* channel is not in the synchronisation set of the parallel composition, and so communications from the pipeline's components through this channel occur independently: the results are interleaved. In fact, in spite of this interleaving, the ascending order required by *Primes* is preserved. Each filter stage of the pipeline outputs on *out* only after it has received its first communication from its predecessor, but before its first communication to its successor. Together with the initial and final behaviours of *Start* and *Finish*, this forces the strict sequencing of the output. Interference on state components is not a problem in the above parallel compositions, since the state of a process is encapsulated.

Proving that *Primes* is refined by *Eratosthenes* is not within the scope of this paper. Nonetheless, the technique put forward in [16, 17] can be used to calculate *Eratosthenes*' behaviour from *Primes* in the refinement calculus style. The semantics presented in this paper is a foundation for that technique.

3 Unifying Theories of Programming

The semantics of *Circus* [8, 21] is based on the unifying theory, but with the Z notation used as the concrete syntax for the relational calculus; thus, a *Circus* program denotes a Z specification. The use of Z is not essential to our approach, but it is convenient as Z is well-suited to the definition of relations, has a precise semantics, and has supporting tools.

In the unifying theory, Tarski's relational calculus is used to give a denotational semantics to constructs taken from different programming paradigms. This common framework allows connections to be established between different paradigms and between a theory and its implementable subtheories. Specifications, designs, and programs are all interpreted as relations between an initial and a subsequent observation of a computing device. Distinguished variables are used to describe relevant observations. The relations are defined as predicates over observational variables and their dashed counterparts; they represent the corresponding values before and after the observation.

In accordance with the philosophy of the unifying theory, *Circus* brings together Z, CSP, and the refinement calculus in a language with a single, coherent semantics. The observational variables describe stability from divergence (*okay*), termination (*wait*), a history of interaction with the environment (*tr*), and a set of refused events (*ref*). Together with program variables, these observational variables and their associated healthiness conditions define the subtheory of imperative, communicating, sequential processes and designs. The result is a state-based expression of the failures-divergences model with embedded imperative features.

Because their semantics are so close, *Circus* can be used as a development method for occam programs. Hoare and Roscoe use a style similar to the unifying theory in their semantics for occam [22]: the meaning of a program is given as a predicate over the values of computations, traces, refusals, and a state variable that records termination and divergence in a similar way to our *okay* and *wait* variables.

The unifying theory includes a relational definition of the weakest precondition of a sequential mechanism; that definition, however, is not valid for reactive systems. This is actually not surprising, since in the context of sequential programs, weakest preconditions are concerned with final and initial states only, but in a reactive system, intermediate states are also relevant. It is in these intermediate states that a reactive process is waiting for interaction with its environment. This motivates our generalisation; the same sort of issue is also discussed by Lamport [23].

In the next section, we propose the *weakest reactive precondition* of a relation in the subtheory used to model *Circus*. This characterises the weakest precondition that guarantees that a given condition holds in every observable state: either final or not. Refinement in terms of weakest reactive preconditions is as usual for predicate transformers, and is equivalent to that in the unifying theory.

4 Weakest Reactive Precondition Semantics

The set of channels in scope is relevant to the semantics of the actions. We record them in a channel environment defined as follows.

$$ChanEnv == ChanName \twoheadrightarrow \mathsf{Expression}$$

As already mentioned, as a meta-language, we use Z with a few extensions that we explain as they arise. A channel environment associates a channel name, an element of the given set *ChanName*, to a Z expression that gives its type.

The relational semantics of actions is given by the function $[\![\_]\!]^{\mathcal{A}}$.

$$[\![\_]\!]^{\mathcal{A}} : \mathsf{Action} \rightarrow ChanEnv \rightarrow \mathsf{N} \rightarrow \mathsf{Schema\text{-}Exp}$$

It takes as extra parameters a channel environment and the name of the schema that defines the user state. It gives as result a schema over the observational variables: the components of the process and the user states.

The process state is defined as follows.

$$ProcessState \mathrel{\widehat=} [\, tr : \mathrm{seq}\, Event;\ \ ref : \mathbb{P}\, Event;\ \ okay, wait : Bool \,]$$

Event is a free type determined by the channels in scope: for each such channel *c*, we have a constructor c that takes a value of the type of *c* to an *Event*. We also use a free type *Bool* to model *true* and *false*; boolean variables as used as predicates, for simplicity.

Changes over the process state can only increase the trace of events.

$$ProcStateObs \mathrel{\widehat=} [\, \Delta ProcessState \mid tr \ \mathsf{prefix}\ tr' \,]$$

Other restrictions over the state are more conveniently enforced by the semantic definitions and are discussed later in this section.

The semantics of actions are operations over a state that also includes the components of the user state in the process description.

$$State \mathrel{\widehat=} UserState \wedge ProcessState$$

Here, we assume that the user state is *UserState*. A change in *State* is a process observation.

$$ProcObs \mathrel{\widehat=} \Delta UserState \wedge ProcStateObs$$

Both the restriction above on changes to the process state and any existing restrictions on changes to the user state are enforced.

Actually, we consider families of schemas $ProcObs(USt)$ and $State(USt)$: one for each user state *USt*. We need this generalisation because the user state can be extended by input and local variables, and by parameter declarations.

The definition of $[\![_]\!]^{\mathcal{A}}$is as follows.

Definition 1

$$[\![A]\!]^{\mathcal{A}} \gamma\ USt \ = \ [\![A]\!]^{\mathcal{A_N}} \gamma\ USt \lor Diverge(USt) \lor Wait(USt)$$

The behaviour of A in the situation where the previous operation has not diverged and has not terminated, or rather, in a state where *okay* and $\neg$ *wait* hold, is characterised by $[\![\_]\!]^{\mathcal{A_N}}$. This function takes the same arguments as $[\![\_]\!]^{\mathcal{A}}$, and also gives a schema as result. It is further discussed later in this section.

The family of schemas $Diverge(USt)$ gives the semantics when *okay* is false.

$$Diverge(USt) \mathrel{\widehat=} [\, ProcObs(USt) \mid \neg\ okay\,]$$

In this case, there are no guarantees as the previous operation diverged.

Finally, if the previous operation has not finished, A behaves as follows.

$$Wait(USt) \mathrel{\widehat=} [\, \Xi State(USt) \mid okay \land wait\,]$$

The state is not changed by A, because its execution has not started yet.

In this work, we present a new semantic function $[\![\_]\!]^{\mathcal{WP}}$ for actions.

Definition 2

$$[\![\_]\!]^{\mathcal{WP}} : \mathsf{Action} \mathrel{+\!\!\!\!\to} ChanEnv \mathrel{+\!\!\!\!\to} \mathsf{N} \mathrel{+\!\!\!\!\to} \mathsf{Schema\text{-}Exp} \to \mathsf{Schema\text{-}Exp}$$

$$[\![A]\!]^{\mathcal{WP}}\ \gamma\ USt\ \varphi \mathrel{\widehat=} [\, State(USt);\ Inp \mid wrp_{USt}.[\![A]\!]^{\mathcal{A}}.\varphi\,]$$

where

$$wrp_{USt}.p.\psi \mathrel{\widehat=} \forall\, State(USt)' \bullet p \Rightarrow \psi$$

and Inp is the declaration of any input variables in scope for A.

This function gives the weakest reactive precondition of A, with conditions expressed using schemas. It takes a channel environment γ and a user state name *USt* as arguments. It also takes a schema φ to yield another schema expressing the situations in which φ holds in all subsequent states of A: both intermediate and final. The first schema is a relation on *ProcObs*, and the second defines a restriction on $State(USt)$ and any input variables *Inp* in scope.

The predicate in the schema $[\![A]\!]^{\mathcal{WP}}\ \gamma\ USt\ \psi$ is an application of the function wrp_{USt} to the relational model $[\![A]\!]^{\mathcal{A}}$ of A and ϕ. For historical reasons, we use the dot notation for application of the *wrp* function, rather than the relational notation used in the unifying theories work or the Z notation. In the sequel, for brevity, we omit the parameter *USt* when it is clear from the context.

The function *wrp* takes two predicates as arguments; in Definition 2, we are using the schemas $[\![A]\!]^{\mathcal{A}}$ and φ as predicates, a usual practice in Z. For predicates p and ψ, the weakest reactive precondition for p to establish ψ, or rather $wrp.p.\psi$, is that, in all subsequent states, which are characterised by the dashed state components, if p holds, so does ψ.

We can deduce the following from Definitions 2 and 1.

Theorem 1

$$wrp.[\![A]\!]^{\mathcal{A}} \gamma\ USt.\psi = wrp.[\![A]\!]^{\mathcal{A_N}} \gamma\ USt.\psi \land wrp.Diverge.\psi \land wrp.Wait.\psi$$

This means that we provide a weakest reactive precondition model for an action A by considering the weakest reactive precondition of *Diverge*, of *Wait*, and of the relational model

$[\![A]\!]^{\mathcal{A}_\mathcal{N}}$ of A separately. For *Diverge*, we have the following.

Theorem 2

$$wrp.Diverge.\psi = (\neg\ okay \Rightarrow \forall\ State' \bullet tr \ \mathsf{prefix}\ tr' \Rightarrow \psi)$$

This means that, in the presence of divergence, $\neg\ okay$, whatever property ψ we want to ensure has to be valid under only the assumption that the trace is extended: $tr\ \mathsf{prefix}\ tr'$.

For *Wait*, the result is as follows. The function α gives the set of components of a given schema. Below, we make use of a slight abuse of notation and write $\psi[\alpha State/\alpha State']$ to mean substitution in ψ of every state component for the corresponding dashed one.

Theorem 3

$$wrp.Wait.\psi = (okay \wedge wait \Rightarrow \psi[\alpha State/\alpha State'])$$

If the previous operation has not finished, $okay \wedge wait$, A cannot establish any property that does not already hold, as it cannot start.

The theorem below provides a way of calculating $wrp.p.\psi$, for an arbitrary ψ, in such a way that $wrp.p$ is actually applied to a predicate that does not involve undashed variables.

Theorem 4 *For a list cl of fresh constants,*

$$wrp.p.\psi = (wrp.p.\psi[cl/\alpha State])[\alpha State/cl]$$

First, the undashed variables of ψ are replaced with fresh constants cl; next $wrp.p$ is calculated for this new predicate $\psi[cl/\alpha State]$; and afterwards, the undashed variables are restored. Based on this theorem, we calculate $wrp.[\![A]\!]^{\mathcal{A}_\mathcal{N}}\gamma\ USt.\psi$ only for predicates ψ that do not involve undashed variables; we call these predicates conditions. The proof of this theorem is a simple application of predicate calculus and substitution properties.

In the definition of $[\![A]\!]^{\mathcal{A}_\mathcal{N}}\gamma\ USt$, we use the family of schemas below, which characterises the situation in which the previous action has not diverged and has finished, $okay \wedge \neg\ wait$, and so A can proceed.

$$Normal(USt) \mathrel{\widehat=} [\,ProcObs(USt) \mid okay \wedge \neg\ wait\,]$$

For the calculation of $wrp.[\![A]\!]^{\mathcal{A}_\mathcal{N}}\gamma\ USt.\psi$, it is useful to define $wrpn$, the weakest reactive precondition that guarantees that p establishes a condition when it is actually activated.

Definition 3

$$wrpn_{USt}.p.\psi \mathrel{\widehat=} wrp_{USt}.Normal(USt) \wedge p.\psi$$

We are using $Normal(USt)$ above as a predicate.

A simple calculation lets us obtain the following result.

Theorem 5

$$wrpn.p.\psi = okay \wedge \neg\ wait \Rightarrow wrp.tr\ \mathsf{prefix}\ tr' \wedge p.\psi$$

In words, if p can proceed, then ψ has to hold whenever the trace is extended in the way prescribed by p.

4.1 Schema Expressions

The relational semantics of a schema expression is defined as follows.

Definition 4

$$[\![SExp]\!]^{\mathcal{A}_\mathcal{N}} \gamma\ USt = SExp \land OpNormal \lor OpDiverge$$
$$OpNormal \mathrel{\widehat=} [\, Normal(USt) \mid tr' = tr \land okay' \land \neg\ wait' \,]$$
$$OpDiverge \mathrel{\widehat=} [\, Normal(USt);\ SExp \lor \neg\ SExp \mid \neg\ \text{pre}\ SExp \,]$$

When a schema action *SExp* is activated, if its precondition does not hold, then it diverges: no conditions are guaranteed; this is characterised by *OpDiverge* above. In that schema, the inclusion of $SExp \lor \neg\ SExp$ has the sole purpose of bringing any input variables into scope. If, on the other hand, the precondition holds, then the action changes the user state as prescribed in *SExp* and terminates successfully without changing the trace. This is captured by the schema *OpNormal*.

The theorem below gives the weakest reactive precondition of *SExp*. If the precondition does not hold, *SExp* diverges and the required condition ψ has to hold with the only assumption that the trace is increased. If the precondition of *SExp* holds, then ψ has to hold when the trace is not changed, $okay'$ is true and $wait'$ is false, as *SExp* does not communicate any values and terminates. The quantification over ref' means that no restrictions on refusals are guaranteed: *SExp* refuses all communications.

Theorem 6

$$\begin{array}{l} wrp.[\![SExp]\!]^{\mathcal{A}_\mathcal{N}} \gamma\ USt.\psi\ = \\ \quad okay \land \neg\ wait \Rightarrow \\ \quad\quad (\neg\ \text{pre}\ SExp \Rightarrow (\forall\ State' \bullet tr\ \mathsf{prefix}\ tr' \Rightarrow \psi)\,) \land \\ \quad\quad (\forall\ ref' : \mathbb{P}\,Event;\ USt' \bullet SExp \Rightarrow \psi[tr, true, false/tr', okay', wait']\,) \end{array}$$

Proof

$$\begin{array}{ll} wrp.[\![SExp]\!]^{\mathcal{A}_\mathcal{N}} \gamma\ USt.\psi & \\ = wrp.(SExp \land OpNormal \lor OpDiverge).\psi & [\text{Definition 4}] \\ = wrp.SExp \land OpNormal.\psi \land wrp.OpDiverge.\psi & [\text{property of } wrp] \\ = wrpn.SExp \land tr' = tr \land okay' \land \neg\ wait'.\psi \land wrpn.\neg\ \text{pre}\ SExp.\psi & \\ & [\text{definitions of } wrpn,\ OpNormal, \text{ and } OpDiverge] \\ = (okay \land \neg\ wait \Rightarrow & \\ \quad \forall\ ref' : \mathbb{P}\,Event;\ USt' \bullet SExp \Rightarrow \psi[tr, true, false/tr', okay', wait']) \land & \\ \quad (okay \land \neg\ wait \Rightarrow \neg\ \text{pre}\ SExp \Rightarrow \forall\ State' \bullet tr\ \mathsf{prefix}\ tr' \Rightarrow \psi) & \\ & [\text{definitions of } wrpn \text{ and } wrp, \text{ and predicate calculus}] \\ = okay \land \neg\ wait \Rightarrow & \\ \quad (\neg\ \text{pre}\ SExp \Rightarrow (\forall\ State' \bullet tr\ \mathsf{prefix}\ tr' \Rightarrow \psi)\,) \land & \\ \quad (\forall\ ref' : \mathbb{P}\,Event;\ USt' \bullet SExp \Rightarrow \psi[tr, true, false/tr', okay', wait']\,) & \\ & [\text{predicate calculus}] \end{array}$$

□

In the sequel, we omit simple proofs for the sake of conciseness.

Example: We consider again the process *WSum* presented in Section 2. It includes the schema action *Compute* reproduced below.

$$Compute \mathrel{\widehat{=}} [\, \Delta S;\ x? : \mathbb{N} \mid val' = a * last + b * x? \land last' = x? \,]$$

This action updates *val*, so that it records the weighted sum of the input $x?$ and *last* with weights a and b, and updates *last* to record the value of the input.

We calculate the weakest reactive precondition for *Compute* to finish and record as output the value 2.

$$\psi \mathrel{\widehat{=}} [\, \Delta S \mid okay' \land \neg\, wait' \land val' = 2 \,]$$

By definition of $[\![\_]\!]^{\mathcal{WP}}$, we have the following.

$$[\![Compute]\!]^{\mathcal{WP}} \gamma\, S.\psi = [\, S;\ x? : \mathbb{N} \mid wrp.[\![Compute]\!]^{\mathcal{A}}.\psi \,]$$

According to Theorem 1, we can calculate $wrp.[\![Compute]\!]^{\mathcal{A}}.\psi$ in terms of the weakest reactive precondition for $[\![Compute]\!]^{\mathcal{A_N}}$, for *Diverge*, and for *Wait*. We start with *Diverge*.

$$\begin{aligned}
& wrp.Diverge.\psi \\
&= \neg\, okay \Rightarrow \forall\, State' \bullet tr \;\mathsf{prefix}\; tr' \Rightarrow okay' \land \neg\, wait' \land val' = 2 && \text{[Theorem 2]} \\
&= \neg\, okay \Rightarrow false && \text{[predicate calculus]} \\
&= okay && \text{[predicate calculus]}
\end{aligned}$$

This means that *Diverge*, which is characterised by $\neg\, okay$, can only establish ψ if *okay*; therefore, *Diverge* cannot establish the condition itself. Divergence has to be avoided if ψ.

For *Wait*, we proceed as follows.

$$\begin{aligned}
& wrp.Wait.\psi \\
&= okay \land wait \Rightarrow okay \land \neg\, wait \land val = 2 && \text{[Theorem 3]} \\
&= \neg\, okay \lor \neg\, wait \lor (okay \land \neg\, wait \land val = 2) && \text{[predicate calculus]} \\
&= \neg\, (okay \land wait) && \text{[predicate calculus]}
\end{aligned}$$

Even though it is possible for *Wait* to establish $val' = 2$, if *val* is already 2, *Wait* cannot terminate, and ψ requires that it does; so, we also have to avoid $okay \land wait$ to obtain ψ.

Finally, we consider $[\![Compute]\!]^{\mathcal{A_N}}$. We observe that pre $Compute = true$.

$$\begin{aligned}
& wrp.[\![Compute]\!]^{\mathcal{A_N}}.\psi \\
&= okay \land \neg\, wait \Rightarrow && \text{[Theorem 6]} \\
&\qquad (\neg\, true \Rightarrow \forall\, State' \bullet tr \;\mathsf{prefix}\; tr' \Rightarrow \psi) \land \\
&\qquad (\forall\, ref' : \mathbb{P}\, Event;\ S' \bullet val' = a * last + b * x? \land last' = x? \Rightarrow val' = 2) \\
&= okay \land \neg\, wait \Rightarrow a * last + b * x? = 2 && \text{[predicate calculus]}
\end{aligned}$$

In summary, based on Theorem 1, we can conclude the following.

$$\begin{aligned}
& wrp.[\![Compute]\!]^{\mathcal{A}}.\psi \\
&= okay \land \neg\, (okay \land wait) \land (okay \land \neg\, wait \Rightarrow a * last + b * x? = 2) && \text{[from above]} \\
&= okay \land \neg\, wait \land (okay \land \neg\, wait \Rightarrow a * last + b * x? = 2) && \text{[predicate calculus]} \\
&= okay \land \neg\, wait \land a * last + b * x? = 2 && \text{[predicate calculus]}
\end{aligned}$$

To establish ψ, *Compute* must be able to start (*okay* and $\neg\, wait$), and *last* and $x?$ have to be adequate ($a * last + b * x? = 2$).

4.2 CSP Expressions

The weakest reactive precondition of the *CSP* processes *Skip*, *Stop*, and *Chaos* are as follows.

Theorem 7 *For a condition ψ,*

$$\begin{aligned}
& wrp.[\![Skip]\!]^{\mathcal{A}_\mathcal{N}} \gamma\ USt.\psi\ = \\
& \qquad okay \land \neg\ wait \Rightarrow \forall\ ref' : \mathbb{P}\ Event \bullet \psi[tr, true, false, USt/tr', okay', wait', USt'] \\
& wrp.[\![Stop]\!]^{\mathcal{A}_\mathcal{N}} \gamma\ USt.\psi\ = \\
& \qquad okay \land \neg\ wait \Rightarrow \forall\ ref' : \mathbb{P}\ Event \bullet \psi[tr, true, true, USt/tr', okay', wait', USt'] \\
& wrp.[\![Chaos]\!]^{\mathcal{A}_\mathcal{N}} \gamma\ USt.\psi\ =\ (\ okay \land \neg\ wait \Rightarrow \forall\ State' \bullet tr\ \mathsf{prefix}\ tr' \Rightarrow \psi\ )
\end{aligned}$$

Skip is the process that terminates successfully, immediately. The required condition ψ has to hold when the trace and the user state are not changed, and *okay'* is *true* and *wait'* is *false*. All communications are refused, and so *ref'* is universally quantified.

For example, for any event *e*, the weakest reactive precondition of *Skip* with respect to $tr' = tr \frown \langle e \rangle$ is *false*. We also obtain precondition *false* for the conditions $\neg\ okay'$ and *wait'*, because *Skip* cannot change the trace, diverge, or deadlock. On the other hand, the weakest reactive precondition with respect to $val' = 2$ is $okay \land \neg\ wait \Rightarrow val = 2$, since *Skip* does not change the state.

For *Stop*, which deadlocks immediately, the difference is that the condition has to hold when *wait'* is true. Finally, the definition for *Chaos*, which diverges immediately, is similar to that for the case when the precondition of a schema expression action does not hold.

For sequences, we have the following result.

Theorem 8 *For a condition ψ,*

$$\begin{aligned}
& wrp.[\![A;\ B]\!]^{\mathcal{A}_\mathcal{N}} \gamma\ USt.\psi = \\
& \qquad wrp.[\![A]\!]^{\mathcal{A}_\mathcal{N}} \gamma\ USt.(\psi \land (okay' \land \neg\ wait' \Rightarrow (wrp.[\![B]\!]^{\mathcal{A}} \gamma\ USt.\psi)'))
\end{aligned}$$

Usually, in weakest precondition semantics, the semantics of sequence is functional composition, so that the weakest precondition for a sequence to establish a postcondition is the weakest precondition for the first component to establish the weakest precondition for the second component to establish the postcondition.

For weakest reactive preconditions, however, not only the final state is relevant: all intermediate states, including those of *A*, are relevant. Theorem 8 requires that ψ holds during the execution of *A*. Moreover, if *A* finishes ($okay' \land \neg\ wait'$), then the weakest reactive precondition of *B* with respect to ψ also has to hold. Since conditions are given in terms of dashed variables and preconditions in terms of undashed variables, the free variables of $wrp.[\![B]\!]^{\mathcal{A}_\mathcal{N}} \gamma\ USt.\psi$ need to be dashed. For a predicate *p*, we define *p'* to be the predicate obtained by dashing the free occurrences of undashed observational variables in *p*.

A relational definition for the prefixing operator is presented below: sequential composition of a communication with the prefixed action. The communication can be observed before it actually takes place, and afterwards. These observations are described in the sequel by *CWaiting*(*USt*) and *CDone*(*USt*). The communication itself, *Comm*(*USt*), is described by the disjunction of these schemas.

A communication takes as input the set *accE*? of acceptable events. Before the communication takes place, the state and the trace do not change, the acceptable events cannot be refused, and the action does not diverge or terminate. After the communication, the trace is augmented by one of the acceptable events and the action terminates successfully. The state is still not changed.

In the definition of the semantics of prefixing, we use a substitution notation for schemas not directly available in Standard Z, although it is in Z/Eves [24]. For a schema S, with a component v of type T, and an expression e of the same type, we denote by $S[v := e]$ the schema $\exists v : T \bullet [\, v = e \,] \land S$.

For an output prefixing, the set of acceptable events of the communication contains only the event obtained by applying the channel, which is an event constructor, to the output value. For an input prefixing, this is the set of events resulting from applying the channel to all the values of its type, obtained from the channel environment γ.

A communication also outputs the value $e!$ communicated. For an output prefixing, this information is ignored by hiding. For an input prefixing, this information is used to initialise the input variable. Input variables are modelled as extensions to the state; they are local to the prefixing, and so they and their dashed counterparts are quantified.

Definition 5

$$
\begin{array}{|l}
\hline
\mathit{CWaiting}(\mathit{USt}) \\
\mathit{Normal}(\mathit{USt}) \\
\mathit{accE}? : \mathbb{P}\,\mathit{Event} \\
\Xi \mathit{USt} \\
\hline
\mathit{tr}' = \mathit{tr} \land \mathit{accE}? \cap \mathit{ref}' = \emptyset \\
\mathit{okay}' \land \mathit{wait}' \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{CDone}(\mathit{USt}) \\
\mathit{Normal}(\mathit{USt}) \\
\mathit{accE}? : \mathbb{P}\,\mathit{Event} \\
e! : \mathit{Event} \\
\Xi \mathit{USt} \\
\hline
e! \in \mathit{accE}? \\
\mathit{tr}' = \mathit{tr} \frown \langle e! \rangle \\
\mathit{okay}' \land \neg\, \mathit{wait}' \\
\hline
\end{array}
$$

$$\mathit{Comm}(\mathit{USt}) \mathrel{\widehat=} \mathit{CWaiting}(\mathit{USt}) \lor \mathit{CDone}(\mathit{USt})$$

$$[\![c!e \to A]\!]^{\mathcal{A_N}} \gamma\ \mathit{USt} \mathrel{\widehat=} (\mathit{Comm}(\mathit{USt})[\mathit{accE}? := \{\mathtt{c}(e)\}] \setminus e!) \mathbin{;} [\![A]\!]^{\mathcal{A}} \gamma\ \mathit{USt}$$

$$
\begin{array}{l}
[\![c?x \to A]\!]^{\mathcal{A_N}} \gamma\ \mathit{USt} \mathrel{\widehat=} \exists\, x, x' : \gamma\ c \bullet \\
\qquad \mathit{Comm}(\mathit{USt})[\mathit{accE}?, e! := \{ y : \gamma\ c \bullet \mathtt{c}(y) \}, \mathtt{c}(x')] \mathbin{;} [\![A]\!]^{\mathcal{A}} \gamma\ \mathit{USt}
\end{array}
$$

The weakest reactive precondition of the output prefixing operator is given below.

Theorem 9 *For a condition ψ,*

$$
\begin{array}{l}
\mathit{wrp}.[\![c!e \to A]\!]^{\mathcal{A_N}} \gamma\ \mathit{USt}.\psi = \\
\qquad \mathit{wrp}.\mathit{CWaiting}[\mathit{accE}? := \{\mathtt{c}(e)\}].\psi \land \\
\qquad \mathit{wrp}.\mathit{CDone}[\mathit{accE}? := \{\mathtt{c}(e)\}] \setminus e!.(\psi \land (\mathit{wrp}.[\![A]\!]^{\mathcal{A}} \gamma\ \mathit{USt}.\psi)')
\end{array}
$$

The condition has to hold before the communication takes place, after it takes place, and during A, as well. For input prefixing, we have a similar result, but since the input is a local variable, it is universally quantified as usual in weakest precondition semantics.

Example: We consider the action $out!val \rightarrow Skip$, in the context of the process *WSum*, and calculate its weakest reactive precondition with respect to $tr' = tr$.

$$wrp.[\![out!val \rightarrow Skip]\!]^{\mathcal{A}_{\mathcal{N}}} \gamma\ USt.tr' = tr$$

$$\begin{aligned}
= (&wrp.CWaiting[accE? := \{\mathsf{out}(val)\}].tr' = tr_0 \land && \text{[Theorems 4 and 9]}\\
&wrp.CDone[accE? := \{\mathsf{out}(val)\}] \setminus e!.(tr' = tr_0 \land (wrp.[\![Skip]\!]^{\mathcal{A}} \gamma\ USt.tr' = tr_0)')\\
)&[tr/tr_0]
\end{aligned}$$

We use a fresh constant tr_0 to replace the occurrences of tr in ψ. We calculate the weakest reactive precondition of *Skip* with respect to $tr' = tr_0$ separately.

$$wrp.[\![Skip]\!]^{\mathcal{A}} \gamma\ USt.tr' = tr_0$$

$$= wrp.[\![Skip]\!]^{\mathcal{A}_{\mathcal{N}}} \gamma\ USt.tr' = tr_0 \land wrp.Diverge.tr' = tr_0 \land wrp.Wait.tr' = tr_0 \qquad \text{[Theorem 1]}$$

$$= (okay \land \neg\, wait \Rightarrow tr = tr_0) \land okay \land (okay \land wait \Rightarrow tr = tr_0) \qquad \text{[Theorems 7,2, and 3]}$$

$$= okay \land tr = tr_0 \qquad \text{[predicate calculus]}$$

This means that the previous operation must not diverge and the trace has to be already tr_0 for *Skip* to guarantee $tr' = tr_0$.

For the weakest reactive precondition of *CWaiting*, we have the following result.

$$wrp.CWaiting[accE? := \{\mathsf{out}(val)\}].tr' = tr_0$$

$$\begin{aligned}
= \forall\, State' \bullet \qquad && \text{[definitions of } wrp \text{ and } CWaiting\text{]}\\
Normal \land USt' = USt \land tr' = tr \land \{\mathsf{out}(val)\} \cap ref' = \emptyset \land okay' \land wait' \Rightarrow\\
tr' = tr_0
\end{aligned}$$

$$= okay \land \neg\, wait \Rightarrow \forall\, ref' : \mathbb{P}\, Event \bullet \{\mathsf{out}(val)\} \cap ref' = \emptyset \Rightarrow tr = tr_0 \qquad \text{[predicate calculus]}$$

$$= okay \land \neg\, wait \Rightarrow ((\exists\, ref' : \mathbb{P}\, Event \bullet \{\mathsf{out}(val)\} \cap ref' = \emptyset) \Rightarrow tr = tr_0) \qquad \text{[predicate calculus]}$$

$$= okay \land \neg\, wait \Rightarrow tr = tr_0 \qquad \text{[predicate calculus]}$$

This means that the trace has to be already tr_0 for *CWaiting* to establish the condition; *CWaiting* cannot change the trace.

Finally, the weakest reactive precondition of *CDone* can be calculated as follows.

$$wrp.CDone[accE? := \{\mathsf{out}(val)\}] \setminus e!.(tr' = tr_0 \land okay')$$

$$\begin{aligned}
= \forall\, State' \bullet \qquad && \text{[definitions of } wrp \text{ and } CDone\text{]}\\
Normal \land USt' = USt \land tr' = tr \frown \langle \mathsf{out}(val) \rangle \land okay' \land \neg\, wait' \Rightarrow\\
tr' = tr_0 \land okay'
\end{aligned}$$

$$= okay \land \neg\, wait \Rightarrow tr \frown \langle \mathsf{out}(val) \rangle = tr_0 \qquad \text{[predicate calculus]}$$

In words, for *CDone* to guarantee $tr' = tr_0$, we need that tr_0 differs from tr just in that it has an extra element: the output.

The calculation of the weakest reactive precondition of $out!val \to Skip$ can proceed as shown below.

$$wrp.[\![out!val \to Skip]\!]^{\mathcal{A_N}} \gamma\ USt.tr' = tr$$

$$= ((okay \wedge \neg\ wait \Rightarrow tr = tr_0) \wedge \qquad \text{[by the calculations above]}$$
$$(okay \wedge \neg\ wait' \Rightarrow tr ^\frown \langle \mathsf{out}(val) \rangle = tr_0))[tr/tr_0]$$

$$= \neg\ (okay \wedge \neg\ wait) \qquad \text{[predicate calculus]}$$

In summary, if the output prefixing starts, we cannot guarantee that the trace is not changed, as should be expected.

The weakest reactive precondition semantics of a guarded action is rather intuitive.

Theorem 10

$$wrp.[\![p\ \&\ A]\!]^{\mathcal{A_N}} \gamma\ USt.\psi =$$
$$(\,p \Rightarrow wrpn.[\![A]\!]^{\mathcal{A_N}} \gamma\ USt.\psi\,) \wedge (\,\neg\ p \Rightarrow wrp.[\![Stop]\!]^{\mathcal{A_N}} \gamma\ USt.\psi\,)$$

If the guard p holds, then $p\ \&\ A$ behaves as A. Otherwise, it behaves like *Stop*.

The behaviour of a parametrised action $D \bullet A$ is that of A taken in an extended state $DUSt$ that includes the components declared in D. Instantiation fixes the value of these components.

Theorem 11

$$wrp_{USt}.[\![D \bullet A]\!]^{\mathcal{A_N}} \gamma\ USt.\psi = wrp_{DUSt}.[\![A]\!]^{\mathcal{A_N}} \gamma\ USt.\psi$$
$$wrp_{USt}.[\![A(e)]\!]^{\mathcal{A_N}} \gamma\ USt.\psi = (wrp_{USt}.[\![A]\!]^{\mathcal{A_N}} \gamma\ USt.\psi)[e/\alpha D]$$

The function α extracts from a declaration the set of variables it introduces. In $A(e)$, we assume that the parameters of A are given by D. The correspondence between the parameters and the arguments e is positional: in the substitution we use the set αD as the list of the parameters in the order they are declared.

4.3 Commands

The weakest reactive precondition of a specification statement $x : [\,pre, post\,]$ does not insist on termination. The list $\alpha USt \backslash x$ includes the variables of the user state other than x. Similarly, $\alpha USt' \setminus x'$ lists their dashed counterparts.

Theorem 12

$$\begin{array}{l} wrp.[\![x : [pre, post]]\!]^{\mathcal{A_N}} \gamma\ USt.\psi\ = \\ \quad okay \wedge \neg\ wait \Rightarrow \\ \qquad (pre \wedge (\forall\ ref' : \mathbb{P}\ Event;\ x' : X \bullet post \Rightarrow \\ \qquad\qquad\qquad \psi[tr, true, false, (USt \setminus x)/tr', okay', wait', (USt' \setminus x')])) \vee \\ \qquad (\neg\ pre \wedge \forall\ State' \bullet tr\ \mathsf{prefix}\ tr' \Rightarrow \psi) \end{array}$$

where X is the type of x.

Roughly, if the precondition holds, then for all final values of x that satisfy the postcondition,

ψ is required to hold. This is the usual weakest precondition semantics of specification statements. There are, however, a few extra issues. First, no restriction on refusals are guaranteed, since the specification statement terminates. Second, ψ cannot require changes to the trace or to state components other than x; nor can it require divergence or nontermination. Finally, if the precondition does not hold, ψ has to hold under only the assumption that $tr \;\mathsf{prefix}\; tr'$.

The weakest reactive precondition for the miraculous $x : [true, false]$ to establish any condition ψ is $true$, as expected. For abort, $x : [false, true]$, we have

$$okay \land \neg\, wait \Rightarrow (\,\forall\, State' \bullet tr \;\mathsf{prefix}\; tr' \Rightarrow \psi\,)$$

This is the same we obtained for *Chaos*. In other words, the semantics of *Chaos* and abort is the same, as should be expected.

The semantics of an assignment $x := e$ is given by substitution as usual.

Theorem 13

$$\begin{array}{l} wrp.[\![x := e]\!]^{\mathcal{A}_\mathcal{N}}\gamma\; USt.\psi \;\; = \\ \quad okay \land \neg\, wait \Rightarrow \\ \qquad \forall\, ref' : \mathbb{P}\, Event \bullet \\ \qquad\quad \psi[tr, true, false, e', (\alpha USt \setminus x)/tr', okay', wait', x', (\alpha USt' \setminus x')] \end{array}$$

As with specification statements, an assignment refuses all communications, does not change the trace, terminates immediately, and changes no variables other than x.

The semantics of conditionals is as follows.

Theorem 14

$$\begin{array}{l} wrp.[\![\mathbf{if} \;\Box\; i \bullet g_i \rightarrow A_i \;\mathbf{fi}]\!]^{\mathcal{A}}\gamma\; USt \;\; = \\ \quad (\neg\, (\lor\, i \bullet g_i) \Rightarrow wrp.[\![Chaos]\!]^{\mathcal{A}_\mathcal{N}}\gamma\; USt.\psi) \land (\land\, i \bullet g_i \Rightarrow wrpn.[\![A_i]\!]^{\mathcal{A}}\gamma\; USt.\psi) \end{array}$$

If none of the guards is valid, then the conditional aborts: it behaves like *Chaos*. Otherwise, any of the valid guards can be chosen, and so the condition has to be guaranteed for each of the associated actions.

For blocks, we have a standard definition.

Theorem 15

$$wrp_{USt}.[\![\mathbf{var}\; x : T \bullet A]\!]^{\mathcal{A}_\mathcal{N}}\gamma\; USt = \forall\, x : T \bullet wrpn_{xUST}.[\![A]\!]^{\mathcal{A}_\mathcal{N}}\gamma\; USt.\psi$$

The weakest reactive precondition of the body of the block is taken in an extended state *xUSt* that includes the declared variable.

5 Conclusions

The refinement calculus [9, 15] has been influential in understanding the development of sequential programs from their formal specifications. One of the aims of the *Circus* project [6, 7, 8] is to extend this calculational approach to reactive programs.

Since the sequential refinement calculus is based on weakest precondition semantics, in this paper, we have presented a new notion of weakest reactive precondition, *wrp*, to reason about concurrent programs in a similar manner. Based on this new notion and on a relational semantics for *Circus*, we have calculated a weakest reactive precondition semantics for a subset of this language. The notion of refinement in the predicate transformer model is equivalent to that in the unifying theory: the basis of our relational model.

A weakest precondition is an extreme solution to a Hoare triple with one unknown: the precondition. The idea of extending Hoare logic, weakest precondition semantics, and even the refinement calculus to the development of parallel programs has a long pedigree.

The work of Owicki and Gries [25, 26, 27, 28, 29] marks the first significant attempt to extend Hoare logic. Their theory extends Hoare's deductive system for proving the partial correctness of sequential programs [30] by adding parallelism in the form of co-blocks, synchronisation, mutual exclusion, and wait statements.

In their method, parallel processes are considered in isolation and a proof of sequential correctness is obtained. These proofs must then be shown to be free from interference: no wait statement or assignment outside a wait statement in one process interferes with the proof of any other. The specification of the parallel program is then the conjunction of the preconditions and the postconditions of the individual components. An important point is that this method is not compositional, since the verification of each process P requires an interference-freedom proof involving information not in P's specification.

Lamport also generalised Hoare logic for concurrent programs [31]. The Hoare triple $P\,\{\,S\,\}\,Q$ means that if P is true before the execution of S, and S terminates, then Q holds. Lamport changed this meaning in the presence of concurrency: if execution is begun anywhere within S with P true, then P remains true until S terminates, whereupon Q is true. In this way, Lamport suggests that a program is better thought of as an invariance maintainer, rather than as a predicate transformer. Although this technique is compositional, it requires a global invariant, and parallel processes must each have this as their specification. Some independence is gained from structuring this invariant with auxiliary control-flow predicates, but the logic is awkward to use as a method of specifying module interfaces.

In [23], Lamport generalises the weakest liberal precondition and the strongest postcondition to the weakest and the strongest invariant. The method allows the verification of a concurrent program to proceed without knowing the granularity of atomicity of operations, and behavioural arguments to be replaced by assertional reasoning. The difference between those predicate transformers and *wrp* lies in the set of observable states. In our semantics, we consider the relationship between the initial state of a process and any subsequent state, whereas Lamport considers the relationship between any state in which execution is resumed and the subsequent state in which it is suspended.

Back [32] applies the refinement calculus to the stepwise refinement of parallel and reactive programs. Action systems are used as the basic program model: they are sequential programs that can be implemented in a parallel fashion. Reactive programs are data refined using techniques originally developed for the sequential refinement calculus. The main difference between our approach and Back's arises from the fact that *Circus* is based partly on a process algebra, not on an action system. In an action system, control flow is encoded in state information, with variables playing the rôle of program counters; in a process algebra, control flow is described using the process combinators of the algebra. Therefore, our predicate transformers and refinement calculus deals with these combinators.

In our work, we have derived our weakest precondition semantics from an existing relational semantics. As a result, our weakest reactive precondition calculator is very close in spirit to predicate transformers for sequential programs. Moreover, since our model includes refusal sets (the *ref* observational variable), we can reason about liveness as well as safety properties. We are currently seeking to extend our calculations to obtain compositional definitions for some of the operators.

We plan to use this semantic model to justify fully the refinement calculus presented in [16, 17]. We are also starting an effort to propose rules for verification-condition generation based on this work. We plan to develop a series of examples and case studies in the use of *wrp*. Our confidence in the semantics of *Circus* has been greatly increased through this work.

In the unifying theory, the model of reactive programs is further constrained by a number of healthiness conditions to obtain theories for particular mechanisms of synchronisation and communication. We expect that the *wrp* models corresponding to these restricted theories also satisfy a number of healthiness conditions of their own. It is also our intent to further explore these relationships.

Acknowledgments

We would like to thank Augusto Sampaio and the anonymous referees for their inspirational suggestions. This work is partially supported by EPSRC grant GR/R43211/01 on "Refinement calculi for sequential and concurrent programs" and the QinetiQ grant CU009-019346 on "Advances in formal modelling and concurrency as represented by the *Circus* language". Ana Cavalcanti is partly supported by CNPq grant 5207613/98-0 and by Universidade Federal de Pernambuco.

References

[1] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.

[2] J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[3] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[4] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[5] C. Fischer. How to Combine Z with a Process Algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*. Springer-Verlag, 1998.

[6] J. C. P. Woodcock and A. L. C. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, Dublin, Ireland, July 2001.

[7] J. C. P. Woodcock and A. L. C. Cavalcanti. The steam boiler in a unified theory of Z and CSP. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, 2001.

[8] J. .C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of Circus. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184 – 203. Springer-Verlag, 2002.

[9] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[10] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[11] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[12] R. J. R. Back. *On The Correctness of Refinement Steps in Program Development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978. Report A-1978-4.

[13] C. C. Morgan, K. Robinson, and P. H. B. Gardiner. On the Refinement Calculus. Technical Monograph TM-PRG-70, Oxford University Computing Laboratory, Oxford - UK, 1988.

[14] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall, 1990.

[15] A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC - A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267 – 289, 1999.

[16] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in Circus. In *Proceedings of Formal Methods Europe FME'2002*, Lecture Notes in Computer Science. Springer-Verlag, 2002. To appear.

[17] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of Actions in Circus. In *Proceedings of REFINE'2002*, Eletronic Notes in Theoretical Computer Science. Eletronic Notes in Theoretical Computer Science, 2002. Invited paper. To appear.

[18] A. W. Roscoe and C. A.R. Hoare. The Laws of occam Programming. *Theoretical Computer Science*, 60(2):177 – 229, 1988.

[19] A. W. Roscoe. Denotational Semantics for occam. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 306 – 329, 1984.

[20] C. A. R. Hoare. Proof of a structured program: The Sieve of Eratosthenes. *BCS Computer Journal*, 15(4):321 – 325, 1972.

[21] J. C. P. Woodcock and A. L. C. Cavalcanti. *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, 2001.

[22] C. A. R. Hoare and A. W. Roscoe. Programs as executable predicates. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984 (FGCS'84)*, pages 220–228, Tokyo, Japan, November 1984. Institute for New Generation Computer Technology.

[23] L. Lamport. *win* and *sin*: predicate transformers for concurrency. Report 17, Digital Sytems Research Centre, 1989.

[24] I. Meisels. *Software Manual for Windows Z/EVES Version 2.1*. ORA Canada, 2000. TR-97-5505-04g.

[25] S. Owicki and D. Gries. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, 1975.

[26] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319 – 340, 1976.

[27] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach . *Communications of the ACM*, 19(5):279 – 285, 1976.

[28] E. W. Dijkstra. A personal summary of the Gries-Owicki theory. In *Selected writings on computing: a personal perspective*. Springer-Verlag, 1982. EWD554, March 1976.

[29] D. Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921 – 930, 1977.

[30] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576 – 580, 1969.

[31] L. Lamport. The "Hoare Logic" of concurrent programs. *Acta Informatica*, 14:21 – 37, 1980.

[32] R. J. R. Back. Refinement of parallel and reactive programs. In *Proceedings of the Summer School on Program Design Calculi*, Lecture Notes in Computer Science. Springer-Verlag, 1992.