Under consideration for publication in Formal Aspects of Computing

A UTP Semantics for Communicating Processes with Shared Variables and its Formal Encoding in PVS

Ling Shi[1], Yongxin Zhao[2], Yang Liu[3], Jun Sun[4], Jin Song Dong[1,5], and Shengchao Qin[6]

[1]School of Computing, National University of Singapore, Singapore, Singapore
[2]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[3]School of Computer Engineering, Nanyang Technological University, Singapore, Singapore
[4]Singapore University of Technology and Design, Singapore
[5]Institute for Integrated Intelligent Systems, Griffith University, Queensland, Australia
[6]Teesside University, Middlesbrough, UK

Abstract. CSP# (Communicating Sequential Programs) is a modelling language designed for specifying concurrent systems by integrating CSP-like compositional operators with sequential programs updating shared variables. In this work, we define an observation-oriented denotational semantics in an *open* environment for the CSP# language based on the UTP framework. To deal with shared variables, we lift traditional event-based traces into *mixed* traces which consist of state-event pairs for recording process behaviours. To capture all possible concurrency behaviours between action/channel-based communications and global shared variables, we construct a comprehensive set of rules on merging traces from processes which run in parallel/interleaving. We also define refinement to check process equivalence and present a set of algebraic laws which are established based on our denotational semantics. We further encode our proposed denotational semantics into the PVS theorem prover. The encoding not only ensures the semantic consistency, but also builds up a theoretic foundation for machine-assisted verification of CSP# specifications.

Keywords: UTP; Denotational semantics; Shared variables; Encoding

1. Introduction

Communicating Sequential Processes (CSP) [Hoa85], a prominent member of the process algebra family, has been designed to formally model concurrent systems whose behaviours are described as process expressions together with a rich set of compositional operators. It has been widely accepted and applied to a variety of safety-critical systems [WLBF09]. However, with the increasing size and complexity of concurrent systems, it becomes clear that CSP is deficient to model non-trivial data structures (for example, hash tables) or

Correspondence and offprint requests to: Yongxin Zhao, East China Normal University, China. E-mail: yxzhao@sei.ecnu.edu.cn.

functional aspects. To solve this problem, considerable efforts on enhancing CSP with data aspects have been made. One approach is to integrate CSP (or CCS [Mil89], another popular process algebra) with state-based specification languages, such as Circus [WC02], CSP-OZ [Fis97, Smi97], TCOZ [MD00, MD02, QDC03], CSP_σ [CH09], CSP‖B [ST05], and CCS+Z [GS97, TA97].

Inspired by the related works, CSP# [SLDC09] has been proposed to specify concurrent systems which involve *shared variables*. It combines the event-based specifications with the state-based programs by introducing data operations to handle state transitions. CSP# integrates CSP-like compositional operators with sequential program constructs such as assignments and while loops, for the purpose of expressive modelling and efficient system verification. In addition, CSP# is supported by the PAT model checker [SLDP09] and has been applied to a number of systems available at the PAT website (`http://pat.comp.nus.edu.sg/`).

An operational semantics of CSP# was defined by interpreting the behaviour of CSP# models using labelled transition systems (LTS) [SLDC09]. Based on this semantics, model checking CSP# becomes possible. Nevertheless, the proposed operational semantics is not compositional and thus lacks the support of compositional verification of process behaviours. Meanwhile, the model checking method based on the operational semantics is certainly limited by the state explosion problem. In practice, the method can only be used for checking the finite state transition systems. Therefore, there is a need for defining a compositional denotational semantics to explain the notations of the CSP# language and further developing a theorem proving approach to complement the model checking approach for system verification.

In this article, we propose a denotational semantics for CSP# and develop an interactive theorem proving framework to support compositional verification. We firstly present an observation-oriented denotational semantics for the CSP# language based on the UTP [HH98] framework in an *open* environment, where process behaviours can be interfered with by the environment. The proposed semantics not only provides a rigorous meaning of the language, but also allows one to deduce algebraic laws [HHH+87] describing the properties of CSP# processes. To deal with shared variables, we lift traditional event-based traces into *mixed* traces (consisting of state-event pairs) for recording process behaviours. To handle different types of synchronisation in CSP# (i.e., action-based and synchronised handshake), we construct a comprehensive set of rules on merging traces from processes which run in parallel/interleaving. These rules capture all possible concurrent behaviours between event/channel-based communications and global shared variables.

Proving our deduced algebraic laws is important as such proofs can validate the correctness of our proposed CSP# denotational semantics. However, manual proving is tedious, and subtle mistakes or omissions can easily occur at any stage of the proofs. Moreover, a high grade of automated verification of system properties can save much human effort. Therefore, a tool that allows semantics encoding and supports machine-assisted proof is needed. In this work, we further encode the proposed observation-oriented denotational semantics into the Prototype Verification System (PVS) [ORS92], which is an integrated framework for formal specification and verification. PVS is an interactive theorem prover based on classical higher-order logic, similar to other theorem provers such as HOL [GM93], Isabelle [Pau94] and Coq [BBC+96]. Nonetheless, PVS supports a richer type system and provides the ability to define subtypes. In our encoding, we apply the built-in PVS theories for sets to the CSP# semantics, including the encoding of the semantic model, expressions, sequential programs, and CSP# processes. In addition, we use the predefined function `subset?` to represent the refinement relationship, and apply the predefined fixed point theory to formalising recursive processes.

Contribution We highlight our contributions below.

- We propose a denotational semantics for the CSP# language based on the UTP framework, including a semantic model and process semantics. The proposed semantic model deals with both communicating processes and shared variables. It can also model both action-based synchronisation and synchronised handshake over channels. Moreover, our model can be adapted/enhanced to define the denotational semantics for other languages with similar concurrency mechanisms.
- We encode the proposed denotational semantics in the PVS theorem prover. The encoding not only checks the consistency of the semantics, but also provides a framework for developing machine-assisted verification for CSP# specifications. In addition, the proved laws can act as auxiliary reasoning rules to improve verification automation.

The remainder of this article is organised as follows: Section 2 briefly introduces the CSP# language, the UTP framework and the features of PVS. Section 3 constructs the observation-oriented denotational semantics in an open environment based on the UTP framework; healthiness conditions are also defined to

characterise the semantic domain. Section 4 discusses the algebraic laws. Section 5 formalises our denotational semantics in PVS. Section 6 discusses related work. Section 7 concludes this article with possible future work. This article is based on the publication [SZL+13] with a complete denotational semantics of CSP# and the encoding of CSP# denotational semantics in PVS.

2. Preliminaries

In this section, we first introduce CSP# covering its syntax, informal descriptions, and an illustrative example using the Dekker's mutual exclusion algorithm. We next introduce the UTP theory which will be used in Section 3. Finally, we introduce some knowledge of PVS that will be used in the encoding of CSP# denotational semantics.

2.1. The CSP# Language

CSP# [SLDC09] integrates CSP-like compositional operators with sequential program constructs such as assignments and while loops. It directly supports shared variables which are not available in the original CSP [Hoa85]. Shared variables can be updated in sequential programs.

2.1.1. Syntax

A CSP# model may consist of definitions of constants, variables, channels, and processes. A constant is defined by keyword **#define** followed by a name and a value, e.g., **#define** *max* 5. A variable is declared with keyword **var** followed by a name and an initial value, e.g., **var** $x = 2$, and its scope is global. A channel is declared using keyword **channel** with a name, e.g., **channel** *ch*. A process is specified in the form of $Proc(i_1, i_2, \ldots, i_n) = P$, where *Proc* is the process name, $(i_1, i_2, \ldots, i_n)$ is an optional list of process formal parameters and P is a process expression.

$$
\begin{array}{lll}
Program & ::= CSP\#Par^* & \text{– model} \\
CSP\#Par & ::= \textbf{\#define}\ \ cons\ v & \text{– constant} \\
 & \mid\ \textbf{var}\ \ x = v_1 & \text{– variable} \\
 & \mid\ \textbf{channel}\ \ ch & \text{– channel} \\
 & \mid\ ProcDel & \text{– process} \\
ProcDel & ::= Proc(ParDel) = P\ \mid\ Proc = P & \text{– process declaration} \\
ParDel & ::= i\ \mid\ i, ParDel & \text{– parameter declaration} \\
P, Q & ::= Stop\ \mid\ Skip & \text{– primitives} \\
 & \mid\ a \rightarrow P & \text{– event prefixing} \\
 & \mid\ ch!exp \rightarrow P\ \mid\ ch?m \rightarrow Proc(m) & \text{– channel output/input} \\
 & \mid\ \{prog\} \rightarrow P & \text{– data operation prefixing} \\
 & \mid\ [b]P & \text{– state guard} \\
 & \mid\ P \,\Box\, Q \mid P \sqcap Q & \text{– external/internal choices} \\
 & \mid\ P\ ;\ Q & \text{– sequential composition} \\
 & \mid\ P \setminus X_1 & \text{– hiding} \\
 & \mid\ P \parallel_{(X_1, X_2)} Q\ \mid\ P \,|||_{X_2}\, Q & \text{– parallel/interleaving} \\
 & \mid\ N \mid \mu\, N \bullet P & \text{– recursion}
\end{array}
$$

where *cons* is an identifier for a constant whose value is v, x is an identifier for a variable whose initial value is v_1, *ch* is an identifier for a channel, *Proc* is a process name, i is an identifier for a variable denoting a process formal parameter, P and Q are processes, a is an action name, *exp* is an arithmetic expression, m is a bound variable, *prog* is a sequential program updating global shared variables, b is a Boolean expression, X_1 is a set of actions, X_2 is a set of synchronous channel inputs and outputs, and N is an identifier. In

addition, the syntax of *prog* is illustrated as follows.

$$
\begin{array}{lll}
prog ::= skip & & \text{– idle} \\
\quad | \quad abort & & \text{– abort} \\
\quad | \quad x = exp & & \text{– assignment} \\
\quad | \quad prog;\ prog & & \text{– composition} \\
\quad | \quad \mathsf{if}\ b\ \mathsf{then}\ prog\ \mathsf{else}\ prog & & \text{– conditional} \\
\quad | \quad \mathsf{while}\ b\ \mathsf{do}\ prog & & \text{– iteration} \\
exp \ ::= v \mid x \mid exp + exp \mid exp - exp \mid exp * exp & & \\
b \quad ::= true \mid false \mid exp\ \mathsf{op}\ exp \mid \neg b \mid b \wedge b \mid b \vee b & & \\
\quad\quad where\ \mathsf{op} \in \{==,\ !=, <, \leq, >, \geq\} & &
\end{array}
$$

Stop is the process that communicates nothing and *Skip* is the process that terminates successfully. Event prefixing $a \rightarrow P$ engages in action a first and afterwards behaves as process P. In CSP#, channels are *synchronous* and their communications are achieved by a handshaking mechanism[1]. Specifically, a process $ch!exp \rightarrow P$ which is ready to perform an output through ch will be enabled if another process $ch?m \rightarrow Proc(m)$ is ready to perform an input through the same channel ch at the same time, and *vice versa*. The expression exp in $ch!exp$ is evaluated atomically with the occurrence of the output. In process $\{prog\} \rightarrow P$, $prog$ is executed *atomically*, and after that process behaves as P. Process $[b]P$ waits until condition b becomes *true* and then behaves as P. The checking of condition b is conducted atomically with the occurrence of the first event or state transition in P. External choice $P \ \Box\ Q$ is resolved only by the occurrence of a *visible* event or a state transition, and internal choice $P \sqcap Q$ is resolved non-deterministically. Sequential composition $P\ ;\ Q$ behaves as P until P terminates and then behaves as Q. Process $P \setminus X_1$ hides all occurrences of actions in X_1. In process $P \parallel_{(X_1,X_2)} Q$, P and Q run in parallel, and they synchronise on common communication events in X_1 and communications through synchronous channels in X_2. In contrast, in process $P \mid\mid\mid_{X_2} Q$, P and Q run independently (except for communications through synchronous channels in X_2) [SLS+12].

2.1.2. Concurrency

As mentioned earlier, concurrent processes in CSP# can communicate through shared variables, events, or synchronous channels.

Shared variables in CSP# are globally accessible; they can be read and written by different (parallel) processes. Shared variables can be used in guard conditions, sequential programs in data operations, and expressions in the channel outputs; nonetheless, they can only be updated in sequential programs. Furthermore, to avoid any possible data race problem when programs execute atomically, sequential programs from different processes are not allowed to execute simultaneously.

A synchronisation event, which is also called an action, occurs *instantaneously*, and its occurrence may require simultaneous participation by multiple processes. In contrast, a communication over a synchronous channel is two-way between a sender process and a receiver process. Namely, a handshake communication $ch.exp$ occurs when both processes $ch!exp \rightarrow P$ and $ch?m \rightarrow Q(m)$ are enabled simultaneously. We remark that this two-way synchronisation in CSP# [SLS+12] is different from CSP_M where multi-part synchronisation between many sender and receiver processes is allowed [Ros97].

2.1.3. Example

Dekker's algorithm [Dij68] is the first correct solution to the mutual exclusion problem between two processes. It allows two processes to share a single-use resource without conflicts using shared memory for communication only. The mutual exclusion requirement is assured by three aspects: (1) no process will enter its critical section without setting its flag, (2) one process checks the flag of the other process after setting its own, and (3) if both flags are set, the *turn* variable is used to allow only one process to proceed. In our

[1] Asynchronous channels are supported as well but not discussed in this article; and they can be simulated easily with global variables.

CSP# model, we define shared variables to specify concurrency using shared-memory communication.

1. **var** $turn = 0$
2. **var** $flag0 = false$
3. **var** $flag1 = false$
4. $P0 = \{flag0 = true\} \rightarrow (P0Check;\ (css0 \rightarrow cse0 \rightarrow \{turn = 1\} \rightarrow \{flag0 = false\} \rightarrow P0))$
5. $P0Check = [flag1]flag1readT \rightarrow ([turn == 0]turnRead0 \rightarrow P0Check \square [turn\ !\ = 0]turnRead1 \rightarrow \{flag0 = false\} \rightarrow P0Wait) \square [\neg flag1]flag1readF \rightarrow Skip$
6. $P0Wait = [turn == 0]turnRead0 \rightarrow \{flag0 = true\} \rightarrow P0Check \square [turn\ !\ = 0]turnRead1 \rightarrow P0Wait$
7. $P1 = \{flag1 = true\} \rightarrow (P1Check;\ (css1 \rightarrow cse1 \rightarrow \{turn = 0\} \rightarrow \{flag1 = false\} \rightarrow P1))$
8. $P1Check = [flag0]flag0readT \rightarrow ([turn == 1]turnRead1 \rightarrow P1Check \square [turn\ !\ = 1]turnRead0 \rightarrow \{flag1 = false\} \rightarrow P1Wait) \square [\neg flag0]flag0readF \rightarrow Skip$
9. $P1Wait = [turn == 1]turnRead1 \rightarrow \{flag1 = true\} \rightarrow P1Check \square [turn\ !\ = 1]turnRead0 \rightarrow P1Wait$
10. $Dekker = (((\{turn = 0\} \rightarrow (P0\ |||_{\varnothing}\ P1)) \square (\{turn = 1\} \rightarrow (P0\ |||_{\varnothing}\ P1))) \setminus \{turnRead0, turnRead1, flag0readT, flag0readF, flag1readT, flag1readF\}$

In the above model, shared variable *turn* denotes who has a priority to enter the critical section, and shared variables *flag*0 and *flag*1 denote the status of acquiring the resource to enter the critical section of process 0 and 1, respectively. Our model corresponds to the pseudocode of the Dekker's algorithm [Dek] step by step and uses an event to specify the condition checking over a shared variable and a data operation to specify the value updating of the shared variable in [Dek].

CSP# process $P0$ specifies that before entering the critical section, process 0 first sets its flag to be true (capturing *mutual exclusion requirement* (1)) and then checks the other process's flag (capturing *mutual exclusion requirement* (2)) modelled by CSP# process $P0Check$. $P0Check$ specifies two possible behaviours, on one hand, if the other process's flag is also set (captured by condition *flag*1), then the *turn* variable is checked (capturing *mutual exclusion requirement* (3)). Event $flag1readT$ captures the operation on checking the *true* value of the variable *flag*1 in the pseudocode, and condition checking *flag*1 is performed simultaneously with the occurrence of $flag1readT$. If it is the process 0's turn (condition $turn == 0$ being true), it moves to process $P0Check$ for another checking modelled by process recursion. If the turn belongs to the other process (condition $turn\ !\ = 0$ being true), the flag of process 0 is cleared (modelled by data operation $\{flag0 = false\}$) before busy waiting. Process $P0Wait$ models the busy waiting for the turn assigned to process 0. After the turn being assigned, process 0 sets its flag (modelled by data operation $\{flag0 = true\}$), and moves to the checking process $P0Check$. On the other hand, if process 1's flag is not set ($\neg flag1$ at line 5), then process 0 starts to enter the critical section. We specify event $css0$ and $cse0$ at line 4 to represent a critical section start and a critical section end respectively. After exiting its critical section, process 0 passes the turn ($\{turn = 1\}$ at line 4), and releases the resource ($\{flag0 = false\}$ at line 4). The behaviour of process 1 is similar.

Process *Dekker* specifies the whole behaviour of the algorithm where the initial value of *turn* is set non-deterministically, and processes 0 and 1 run concurrently modelled by the interleaving operator $|||_{\varnothing}$. Events $turnRead0, turnRead1, flag0readT, flag0readF, flag1readT, flag1readF$ are hidden by the hiding operator in the process *Dekker* as these events only specify the condition checking and they do not add anything to the meaning of the program. Note that when processes $P0$ and $P1$ run interleaving, sequential programs (e.g., $flag0 = true$ and $flag1 = true$) in $P0$ and $P1$ respectively will not execute simultaneously according to CSP#'s modelling feature.

2.2. UTP Theory

The Unifying Theories of Programming (UTP) [HH98] uses relations as a unifying basis to define denotational semantics for programs across different programming paradigms. For each programming paradigm, programs are generally interpreted as relations between initial observations and subsequent (intermediate or final) observations of the behaviours of their execution. Relations are represented as predicates over observational variables to capture all aspects of program behaviours.

Theories of programming paradigms in the UTP framework are differentiated by their *alphabet*, *signature* and *healthiness conditions*. The alphabet is a set of observational variables recording external observations of the program behaviour. The signature defines the syntax to represent the elements of a theory. The healthiness conditions are a selection of laws identifying valid predicates that characterise a theory.

The observational variables in the alphabet of a theory record the observations that are relevant to program behaviours. Variables of initial observations are undashed, constituting the input alphabet of a relation, and variables of subsequent observations are dashed, constituting the output alphabet of a relation. For example, in the imperative paradigm, variables $x, y, \ldots, z$ record the initial state of program variables, and $x', y', \ldots, z'$ record the final state of program variables. In a theory of reactive processes, the Boolean variable *wait* distinguishes the intermediate observations of a waiting state from the observations of a final state for reactive processes; the Boolean variable *ok* records the stability of program, i.e., whether it is in a stable state or in a divergent state; variable *tr* records the interaction between a process and its environment; *ref* records the set of events that could be refused before the observation.

The signature of a theory is a set of atomic components called *primitives* and *combinators*. The primitives in the signature of relational programming are assignment $x = e$, empty *skip*, top $\top$ for miracle and bottom $\bot$ for abort. The combinators are conditional $P \lhd b \rhd Q$, composition $P \ ; \ Q$, nondeterminism $P \sqcap Q$ and recursion $\mu X \bullet F(X)$. Here, x is a variable in the alphabet, e is an expression, P and Q are predicates describing behaviours of two programs, X is a bound variable standing for a predicate, and F is a monotonic function.

A healthiness condition is associated with observational variables in the alphabet. It is defined by an idempotent function ϕ on predicates. A healthy program represented by predicate P satisfies healthiness condition ϕ if it is a fixed point of ϕ:

$$P = \phi(P).$$

For example, if a program P has not started, the observation of its behaviour is impossible. This can be captured by a healthiness condition $H(P) = ok \Rightarrow P$ requiring that program P satisfies the following equation:

$$P = H(P) \text{ or } P = (ok \Rightarrow P).$$

In the above example, if Boolean variable *ok* is *true*, then program starts and its behaviour is described by predicate P. If *ok* is *false*, then its behaviour is not restricted as predicate $ok \Rightarrow P$ is *true*.

2.3. Prototype Verification System

Prototype Verification System (PVS) [ORS92, COR+95] is an integrated environment for the development and analysis of formal specifications. It combines an expressive modelling language with an interactive prover that has powerful theorem proving capabilities.

The specification language is based on classical typed higher-order logic. Its type system consists of base types such as *Boolean* (`bool`), *integer* (`int`), *real numbers* (`real`) and type constructors for *function types*, *tuple types*, and *record types*. A function type is usually of the form `[D -> R]`, where `D` and `R` are type expressions, denoting the domain and range of the function respectively. Tuple types (also called product types) have the form `[T1, ..., Tn]`, where the `Ti` are type expressions. Projection function `‘i` is used to project the ith element of the tuple. Record types are of the form `[# a1:T1, ..., an:Tn #]`, where the `ai` are called record accessor or fields and the `Ti` are types. For example, a record type `R` consisting of an integer number `x` and a Boolean variable `b` is specified as `R:TYPE = [# x: int, b: bool #]`, given a record `r: VAR R`, its `x`-component is accessed by `r‘x`. The type system of the PVS is augmented with *predicate subtypes*. Subtypes can be specified in two different ways. Given a type `X` and predicate `P` on the elements of `X`, a subtype of `X` with respect to `P` can be specified as either `T: TYPE={x:X|P(x)}` or `T: TYPE = (P)`. The type checking of subtypes is undecidable, and may lead to proof obligations, called *type correctness conditions* (TCCs). Users are required to discharge these TCCs with the assistance of the PVS prover. Another important feature of PVS type system is the provision of abstract datatypes. Familiar data structures of programming languages such as lists and binary trees can be specified in PVS using the abstract datatypes. For example, the following PVS specification declares a list using abstract datatype.

```
list [T: TYPE]: DATATYPE
 BEGIN
  null: null?
  cons(car: T, cdr: list): cons?
 END list
```

To be specific, `list` is parametric in type `T`, and has two constructors `null` and `cons`: `null` takes no arguments and `cons` takes two arguments, where the first is of the type `T` and the second is a list. Two predicates `null?` and `cons?` are recognisers: `null?` holds for exactly those elements of the `list` datatype that are identical to `null`, and `cons?` holds for exactly those elements of the `list` datatype that are constructed using `cons`. Note that two accessors, `car` and `cdr`, correspond to the two arguments of `cons`; they can only be applied to lists which satisfy the `cons?` predicate.

A PVS specification is given as a collection of parameterised *theories*. Each theory may consist of declarations, definitions and formulas. Declarations are used to define types, variables, constants, and so on. Note that *constant declarations* introduce new constants with their associated types and a value optionally, and constants can be functions, relations or the usual (0-ary) constants. PVS supports *recursive definitions*, which are total functions. Hence, it must be ensured that all recursive functions terminate, specified by a *measure* expression. The measure expression follows the `MEASURE` keyword and ends with an optional order relation following a `BY` keyword. The recursive definition generates a *termination TCC* which denotes that the measure function applied to recursive arguments decreases with respect to a well-formed ordering. A proof obligation must be discharged by users. A *formula* can be declared to introduce an axiom using the keyword `AXIOM` and a theorem using the keyword `LEMMA`. Axioms can be referenced by the command `lemma` during proofs. The body of the formula is a Boolean expression. Moreover, PVS supports the *name overloading* which allows the same name from different theories or within a single theory. The collections of theories are organised by means of importings.

The interactive PVS prover [SORSC01] provides a collection of powerful proof commands to perform induction, propositional and equality reasoning, rewriting, model checking and so on. For example, a frequently used powerful proof command is `grind`, which does skolemization, instantiation, simplification, rewriting and applying decision procedures.

3. The Observation-oriented Semantics for CSP#

In this section, we first define the semantic model including the observational variables and healthiness conditions. We then define the meanings of arithmetic and Boolean expressions as well as the denotational semantics of sequential programs. Based on the semantic model and the semantics of expressions and programs, we define the denotational semantics of CSP# processes.

3.1. Semantic Model

The challenge of defining a denotational semantics for CSP# is to design an appropriate model which can cover not only communications but also the shared variable paradigm and satisfy the compositional property. To address the challenge, we blend communication events with states containing shared variables. Namely, we introduce *mixed* traces to record the interactions of processes with the global environment; each trace is a sequence of communication events or (shared variable) state pairs.

3.1.1. Observational Variables

The following variables are introduced in the alphabet of observations of CSP# process behaviours. Some of them (i.e., ok, ok', $wait$, $wait'$, ref, and ref') are similar to those in the UTP theory for CSP [HH98]. The key difference is that the event-based traces in CSP are changed to mixed traces consisting of state-event pairs.

- ok, ok': Boolean describe the stability of a process.
  $ok = true$ records that the process has started in a stable state, whereas $ok = false$ records that the process has not started as its predecessor has diverged.
  $ok' = true$ records that the process has reached a stable state, whereas $ok' = false$ records that the process has diverged.

- $wait$, $wait'$: $\mathsf{Boolean}$ distinguish the intermediate observations of waiting states from the observations of final states.
  $wait = true$ records that the execution of the previous process has not finished, and the current process starts in an intermediate state, while $wait = false$ records that the execution of the previous process has finished and the current process may start.
  $wait' = true$ records that the next observation of the process is in an intermediate state, while $wait' = false$ records that the next observation is in a terminated state.
- ref, ref': $\mathbb{P}\, Event$ denote a set of actions and channel inputs/outputs that can be refused before or after the observation. The set $Event$ denotes all possible actions and channel input/output directions (e.g., $ch?, ch!$). An input direction $ch?$ denotes any input through channel ch, and a channel output direction $ch!$ denotes any output through channel ch. The set Act denotes all possible actions.
- tr, tr': $\mathrm{seq}((\mathsf{S} \times \mathsf{S}^{\perp}) \cup (\mathsf{S} \times \mathsf{E}))$ record a finite sequence of observations (state pairs or communication events) on the interaction of the processes with the global environment.
  - S is the set of all possible mappings (states), and a state $s : \mathsf{VAR} \to \mathsf{int}$ is a total function which maps global shared variables names from VAR into values of integer int. Notice that the types of variable values and channel messages are integer in our proposed semantics.
  - E is the set of all possible events, including actions, channel inputs/outputs and synchronous channel communications.
  - $\mathsf{S} \times \mathsf{S}^{\perp}$ is the set of state pairs, and each pair consists of a pre-state recording the initial variable values before the observation and a post-state recording the final values after the observation. $\mathsf{S}^{\perp} \mathrel{\widehat{=}} \mathsf{S} \cup \{\perp\}$ represents all states, where the improper state $\perp$ indicates non-termination. Note that the state pair is used to record the observation for the data operation.
  - $\mathsf{S} \times \mathsf{E}$ denotes a set of occurring events under the pre-states. The reason of recording the pre-state is that the value of the expression which may contain shared variables in a channel output shall be evaluated under this state.

3.1.2. Healthiness Conditions

Healthiness conditions are defined as equations in terms of an idempotent function ϕ on predicates. Every healthy program represented by predicate P must be a fixed point under the healthiness condition of its respective UTP theory, i.e., $P = \phi\ (P)$.

In CSP#, a process can never change the past history of the observations; instead, it can only extend the record, captured by healthiness condition **R1**. We use predicate P to represent the semantics of the CSP# process below. Predicate $tr \leq tr'$ states that tr is a prefix of tr'.

R1: $\mathbf{R1}(P) \ = \ P \wedge tr \leq tr'$

The execution of a process is independent of the history before its activation, captured by function **R2**.

R2: $\mathbf{R2}(P(tr, tr')) \ = \ \sqcap_s\, P(s, s \frown (tr' - tr))$

As mentioned earlier, variable $wait$ distinguishes a waiting state from the final state. A process cannot start if its predecessor has not finished, or otherwise, the values of all observational variables are unchanged, characterised by function **R3**.

R3: $\mathbf{R3}(P) \ = \ II \lhd wait \rhd P$

where $P \lhd b \rhd Q \mathrel{\widehat{=}} b \wedge P \vee \neg b \wedge Q$ and $II \ \mathrel{\widehat{=}} \ (\neg ok \wedge tr \leq tr') \vee (ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref)$. Here II states that if a process is in a divergent state, then only the trace can be extended, or otherwise, it is in a stable state, and the values of all observational variables remain unchanged.

When a process is in a divergent state, it can only arbitrarily extend the trace. This feature is captured by function **CSP1**.

CSP1: $\mathbf{CSP1}(P) \ = \ (\neg ok \wedge tr \leq tr') \vee P$

Every process is monotonic in the observational variable ok'. This monotonicity property is modelled by function **CSP2** which states that if an observation of a process is valid when ok' is false, then the observation should also be valid when ok' is true.

CSP2: $\mathbf{CSP2}(P) \ = \ P \ ; \ ((ok \Rightarrow ok') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref)$

We below use **H** to denote all healthiness conditions satisfied by the CSP# process.

$\mathbf{H} = \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3} \circ \mathbf{CSP1} \circ \mathbf{CSP2}$

From the above definition, we can see that although CSP# satisfies the same healthiness conditions of CSP, observational variables tr, tr' in our semantic model record additional information for shared variable states. We adopt the same names for the idempotent functions used in CSP for consistency. In addition, function **H** is idempotent and monotonic [CW06, HH98].

3.2. Semantics of Expressions and Programs

In this section, we define the semantics of arithmetic expressions, Boolean expressions and programs. The definitions will be used in Section 3.3.

Definition 1 (Arithmetic Expression). Let Aexp be the set of arithmetic expressions defined in Section 2.1.1 ($exp \in \mathsf{Aexp}$), the evaluation of the expression is defined as a function $\mathcal{A} : \mathsf{Aexp} \to (\mathsf{S} \to \mathsf{int})$.

$$
\begin{array}{ll}
\mathcal{A}[\![v]\!](s) & = v \\
\mathcal{A}[\![x]\!](s) & = s(x) \\
\mathcal{A}[\![exp_1 + exp_2]\!](s) & = \mathcal{A}[\![exp_1]\!](s) + \mathcal{A}[\![exp_2]\!](s) \\
\mathcal{A}[\![exp_1 - exp_2]\!](s) & = \mathcal{A}[\![exp_1]\!](s) - \mathcal{A}[\![exp_2]\!](s) \\
\mathcal{A}[\![exp_1 * exp_2]\!](s) & = \mathcal{A}[\![exp_1]\!](s) * \mathcal{A}[\![exp_2]\!](s)
\end{array}
$$

Definition 2 (Boolean Expression). Let Bexp be the set of Boolean expressions defined in Section 2.1.1 ($b \in \mathsf{Bexp}$), given a valuation, function $\mathcal{B}$ returns whether a Boolean expression is valid, defined as $\mathcal{B} : \mathsf{Bexp} \to (\mathsf{S} \to \mathsf{Boolean})$.

$$
\begin{array}{ll}
\mathcal{B}[\![true]\!](s) & = true \\
\mathcal{B}[\![false]\!](s) & = false \\
\mathcal{B}[\![exp_1 \ \mathsf{op} \ exp_2]\!](s) & = \begin{cases} true & \text{if } \mathcal{A}[\![exp_1]\!](s) \ \mathsf{op} \ \mathcal{A}[\![exp_2]\!](s), \text{ where } \mathsf{op} \in \{==, !=, <, \leq, >, \geq\} \\ false & \text{otherwise} \end{cases} \\
\mathcal{B}[\![\neg b]\!](s) & = \neg(\mathcal{B}[\![b]\!](s)) \\
\mathcal{B}[\![b_1 \wedge b_2]\!](s) & = \mathcal{B}[\![b_1]\!](s) \wedge \mathcal{B}[\![b_2]\!](s) \\
\mathcal{B}[\![b_1 \vee b_2]\!](s) & = \mathcal{B}[\![b_1]\!](s) \vee \mathcal{B}[\![b_2]\!](s)
\end{array}
$$

Definition 3 (Sequential Program). Let Prog be the set of sequential programs defined in Section 2.1.1 ($prog \in \mathsf{Prog}$), function $\mathcal{C}$ returns the updated valuations after executing the program, defined as $\mathcal{C} : \mathsf{Prog} \to (\mathcal{S} \times \mathcal{S}^{\perp})$.

$$
\begin{array}{ll}
\mathcal{C}[\![skip]\!] & = \{(s, s) \mid s \in \mathcal{S}\} \\
\mathcal{C}[\![abort]\!] & = \{(s, s') \mid s \in \mathcal{S}, s' \in \mathcal{S}^{\perp}\} \\
\mathcal{C}[\![x = exp]\!] & = \{(s, s[\mathcal{A}[\![exp]\!](s)/x]) \mid s \in \mathcal{S}\} \\
\mathcal{C}[\![prog_1;\ prog_2]\!] & = \{(s, s') \mid \exists\, s_0 \in \mathcal{S} \bullet (s, s_0) \in \mathcal{C}[\![prog_1]\!] \wedge (s_0, s') \in \mathcal{C}[\![prog_2]\!]\} \cup \\
 & \quad \{(s, \perp) \mid (s, \perp) \in \mathcal{C}[\![prog_1]\!]\} \\
\mathcal{C}[\![\mathsf{if}\ b\ \mathsf{then}\ prog_1\ \mathsf{else}\ prog_2]\!] & = \{(s, s') \mid \mathcal{B}[\![b]\!](s) \wedge (s, s') \in \mathcal{C}[\![prog_1]\!]\} \cup \\
 & \quad \{(s, s') \mid \neg(\mathcal{B}[\![b]\!](s)) \wedge (s, s') \in \mathcal{C}[\![prog_2]\!]\} \\
\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ prog]\!] & = \{(s, s') \mid (s, s') \in \mathcal{C}[\![\mu\, X \bullet F(X)]\!]\}
\end{array}
$$

In the above definition, continuous function $F(X) \mathrel{\widehat{=}} \mathsf{if}\ b\ \mathsf{then}\ prog;\ X\ \mathsf{else}\ skip$, and $\mathcal{C}[\![\mu\, X \bullet F(X)]\!] \mathrel{\widehat{=}} \bigcap_n \mathcal{C}[\![F^n(abort)]\!]$.

3.3. Semantics of Processes

In this section, we construct an observation-oriented semantics for all CSP# process operators based on our proposed UTP semantic model for CSP#. We define the semantics in an open environment to achieve the compositionality property; namely, a process may be interfered with by the environment. In Section 3.1.1, we defined a mixed trace to record the potential events and state transitions in which a process P may engage; for example, the trace $tr' = \langle (s_1, s_1'), (s_2, a_2) \rangle$ describes the transitions of process P. In an open environment, tr' may contain an (implicit) transition (s_1', s_2) as the result of interference by the environment where states s_1' and s_2 can be different.

In the following, we illustrate our semantic definitions for the CSP# process operators, and present the refinement definition.

3.3.1. Primitives

Deadlock process *Stop* never engages in any event or updates shared variables, and it is always waiting.

$$Stop \mathrel{\widehat{=}} \mathbf{H}(ok' \wedge tr' = tr \wedge wait')$$

The semantics shows that the trace is unchanged and process is in a waiting state (represented by $wait'$ being true). In addition, *Stop* refuses all events, so the final value of the refusal set, ref', is left unconstrained.

Process *Skip* terminates immediately without any event or state change occurring.

$$Skip \mathrel{\widehat{=}} \mathbf{H}(\exists\, ref \bullet II)$$

Reactive identity II constrains that if a process terminates, then there is no change on the trace. The initial refusal of *Skip* is irrelevant to its behaviour, defined by the existential quantifier. After termination, the refusal set ref' is arbitrary.

3.3.2. Sequential Composition

In process $P\ ;\ Q$, P takes control first and Q starts only when P has finished.

$$P\ ;\ Q \mathrel{\widehat{=}} \exists\, obs_0 \bullet (P[obs_0/obs'] \wedge Q[obs_0/obs])$$

The semantics of sequential composition shows that if process P diverges, then so does the process $P\ ;\ Q$. If process P is in a waiting state, then the following process Q cannot start. If P terminates, then process Q starts immediately and the final observation of process P is the initial observation of process Q. In the above definition, the term *obs* represents the set of observational variables *ok*, *wait*, *tr*, and *ref*, as is the case for obs_0 and obs'.

3.3.3. Event Prefixing

Process $a \rightarrow P$ engages in event a first and afterwards behaves as process P. Event a defined here is an action which occurs instantaneously, and may require simultaneous participation by more than one processes.

$$a \rightarrow P \mathrel{\widehat{=}} \mathbf{H}\left(ok' \wedge \left(\begin{array}{l} a \notin ref' \wedge tr' = tr \\ \lhd wait' \rhd \\ \exists\, s \in \mathsf{S} \bullet tr' = tr \frown \langle (s, a) \rangle \end{array} \right) \right) ;\ P$$

The above semantics shows two possible behaviours: when a process is waiting to engage in action a, it cannot refuse this action during the waiting period (represented by predicate $a \notin ref'$), and its trace is unchanged; or a process performs action a and terminates with its trace extended with this observation (by predicate $tr' = tr \frown \langle (s, a) \rangle$). Since the environment may interfere with the process behaviour and make a transition on the shared variable states, we use state s from the variable state set S to denote the initial state before the observation. Note that the semantics of sequential composition "; " is defined in Section 3.3.2.

3.3.4. Synchronous Channel Output/Input

In CSP#, messages can be sent/received synchronously through channels. The synchronisation is pairwise, involving two processes. Specifically, a synchronous channel communication *ch.exp* can take place only if an output $ch!exp$ is enabled and a corresponding input $ch?m$ is also ready.

$$ch!exp \rightarrow P \mathrel{\widehat{=}} \mathbf{H}\left(ok' \wedge \left(\begin{array}{l} ch? \notin ref' \wedge tr' = tr \\ \lhd wait' \rhd \\ \exists\, s \in \mathsf{S} \bullet tr' = tr \frown \langle (s, ch!\mathcal{A}[\![exp]\!](s)) \rangle \end{array} \right) \right) ;\ P$$

The above semantics of synchronous channel output depicts two possible behaviours: when a process is waiting to communicate on channel *ch*, it cannot refuse any channel input over *ch* provided by the environment to perform a channel communication (represented by predicate $ch? \notin ref'$), and its trace is unchanged; or a

process performs the output through ch and terminates without divergence. The observation of the trace is recorded as a tuple $(s, ch!\mathcal{A}[\![exp]\!](s))$, where the value of the output message is evaluated under the pre-state s. Here function $\mathcal{A}$ defines the semantics of arithmetic expressions, and its definition is in Definition 1. After the output occurs, the process behaves as P.

$$ch?m \rightarrow Proc(m) \mathrel{\widehat{=}} \exists\, v \in \mathsf{int} \bullet \left(\mathbf{H} \left(ok' \wedge \left(\begin{array}{l} ch! \notin ref' \wedge tr' = tr \\ \lhd wait' \rhd \\ \exists\, s \in \mathsf{S} \bullet tr' = tr \frown \langle (s, ch?v) \rangle \end{array} \right) \right) ;\ Proc(v) \right)$$

As shown above, the semantics of synchronous channel input is similar to channel output except that when a process is waiting, it cannot refuse any channel output provided by the environment, and after the process receiving a message v from channel ch, its trace is appended with a tuple $(s, ch?v)$. In addition, parameter m cannot be modified in process $Proc$; namely, it becomes constant-like and its value is replaced by value v.

3.3.5. Data Operation Prefixing

In process $\{prog\} \rightarrow P$, the sequential program $prog$, which is executed atomically, is called a data operation. The observation of the data operation is the updates on shared variables which are observed after the execution of all programs as illustrated below.

$$\{prog\} \rightarrow P \mathrel{\widehat{=}} \mathbf{H} \left(ok' \wedge \exists\, s \in \mathsf{S} \bullet \left(\begin{array}{l} wait' \wedge tr' = tr \frown \langle (s, \bot) \rangle \\ \lhd (s, \bot) \in \mathcal{C}[\![prog]\!] \rhd \\ \neg wait' \wedge \exists\, s' \in \mathsf{S} \bullet (tr' = tr \frown \langle (s, s') \rangle \\ \qquad\qquad \wedge (s, s') \in \mathcal{C}[\![prog]\!]) \end{array} \right) \right) ;\ P$$

If the evaluation of the program does not terminate (represented by predicate $(s, \bot) \in \mathcal{C}[\![prog]\!]$), then the process is in a waiting state, and its trace is extended with the record of non-termination. On the other hand, if the evaluation succeeds and terminates, then the process terminates and the state transition is recorded in the trace. Note that post-state s' after the observation is associated with the pre-state s under the semantics of sequential programs ($(s, s') \in \mathcal{C}[\![prog]\!]$). Function $\mathcal{C}$ defines the semantics of the sequential programs by inductive definition [Win93], and its definition is in Definition 3. After the data operation occurs, the process behaves as P.

3.3.6. Choice

Internal choice denotes that process $P \sqcap Q$ behaves like either P or Q. The selection is made internally and non-deterministically, not affected by the environment.

$$P \sqcap Q \mathrel{\widehat{=}} P \vee Q$$

External choice denotes that for process $P \,\Box\, Q$, the selection of process P or Q is controlled by the environment, i.e., the choice is resolved by the occurrence of the first visible event or the first state transition.

$$P \,\Box\, Q \mathrel{\widehat{=}} \mathbf{H}((P \wedge Q) \lhd Stop \rhd (P \vee Q))$$

The above definition shows that if no observation has been made and termination has not occurred (i.e., process $Stop$ is true), then the process has both possible behaviours of P and Q. Alternatively, if an observation had been made (i.e., process $Stop$ is false), then process behaviour will be either that of P or that of Q depending on from which choice is made.

3.3.7. State Guard

Process $[b]P$ waits until condition b becomes true and then behaves as P. The checking of condition b is performed simultaneously with the occurrence of the first event or state transition of process P. But note that in some situation, the process P behaves like $Skip$; that is, no state could be observed to judge the truth of condition b. In order to deal with this issue, we construct process $\widehat{P}$ from P by adding a stuttering step. Obviously, all possible difference $(tr' - tr)$ of $\widehat{P}$ are non-empty traces. Thus we can always judge the truth of b according to the behaviours of $\widehat{P}$.

$[b]P \mathrel{\widehat{=}} \widehat{P} \lhd (\mathcal{B}(b)(\pi_1(head(tr' - tr))) \wedge tr < tr') \rhd Stop$
$\widehat{P} \mathrel{\widehat{=}} P \wedge tr < tr' \vee P(tr, tr) \wedge \exists\, s \in \mathsf{S} \cdot tr' - tr = \langle (s, s) \rangle$

The semantics states that if the Boolean guard b is satisfied under the state from the initial observation of $\widehat{P}$, represented by $\pi_1(head(tr' - tr))$, then the observation of whole process is the same as $\widehat{P}$, or otherwise, process behaves as process *Stop*. Function π_1 selects the first element of a tuple and *head* returns the first element of a sequence. Note that the semantics of traditional conditional choice $\mathsf{if}\ (b)\ \{P\}\ \mathsf{else}\ \{Q\}$ can be equivalent to the semantics of $[b]P \vee [\neg b]Q$.

3.3.8. Parallel Composition

The parallel composition $P \parallel_{(X_1, X_2)} Q$ executes P and Q in the following way: (1) common actions of P and Q require simultaneous participation, (2) synchronous channel output in one process occurs simultaneously with the corresponding channel input in the other process, and (3) other events of processes occur independently.

In CSP, the semantics of parallel composition is defined in terms of the merge operator $\parallel_M$ in UTP [HH98], where the predicate M captures how to merge two observations. To deal with channel-based communications and shared variable updates in CSP#, we here define a new merge predicate $M(X_1, X_2)$ to model the merge operation. The set X_1 defined in Section 2.1.1 contains common actions of both processes and the set X_2 all synchronous channel inputs and outputs. Namely,

$$P \parallel_{(X_1, X_2)} Q \mathrel{\widehat{=}} \mathbf{H}\left(\begin{array}{l} \exists\, 0.ok, 0.wait, 0.ref, 0.tr, \\ \quad 1.ok, 1.wait, 1.ref, 1.tr \end{array} \bullet \left(\begin{array}{l} P[0.ok, 0.wait, 0.ref, 0.tr / ok', wait', ref', tr'] \wedge \\ Q[1.ok, 1.wait, 1.ref, 1.tr / ok', wait', ref', tr'] \wedge \\ M(X_1, X_2) \end{array} \right) \right)$$

where

$$M(X_1, X_2) \mathrel{\widehat{=}} \left(\begin{array}{l} (ok' = 0.ok \wedge 1.ok) \wedge \\ (wait' = 0.wait \vee 1.wait) \wedge \\ (ref' = (0.ref \cap 1.ref \cap X_2) \cup ((0.ref \cup 1.ref) \cap X_1) \\ \qquad \cup ((0.ref \cap 1.ref) - X_1 - X_2)) \\ (tr' - tr \in (0.tr - tr \parallel_{X_1} 1.tr - tr)) \end{array} \right)$$

The predicate $M(X_1, X_2)$ captures four kinds of behaviours of a parallel composition. First, the composition diverges if either process diverges (represented by predicate $ok' = 0.ok \wedge 1.ok$). Second, the composition terminates if both processes terminate ($wait' = 0.wait \vee 1.wait$). Third, the composition refuses synchronous channel outputs/inputs that are refused by both processes ($0.ref \cap 1.ref \cap X_2$), all actions that are in the set X_1 and refused by either process ($(0.ref \cup 1.ref) \cap X_1$), and events that are not in the set X_1 and X_2 but refused by both processes ($(0.ref \cap 1.ref) - X_1 - X_2$). Last, the trace of the composition is a member of the set of traces produced by the *trace synchronisation* function $\parallel_{X_1}$ as elaborated below.

Function $\parallel_{X_1}$ models how to merge two individual traces into a set of all possible traces; there are five cases covering both traces are empty, one of the trace is empty and both traces are non-empty. In the following definitions, s_1, s_1', s_2, s_2' are representative elements of variable states with termination, a, a_1, a_2 are representative elements of actions, ch is a representative element of channel names, and v, v_1, and v_2 are values with integer type.

- Firstly, function $\parallel_{X_1}$ is symmetric, i.e., $t_1 \parallel_{X_1} t_2 = t_2 \parallel_{X_1} t_1$.
- The first case covers two scenarios, (1) if both input traces are empty, the result is a set of an empty sequence; (2) if only one input trace is empty, the result is determined based on the first observation of that non-empty trace: if that observation is an action in the set X_1 which requires synchronisation, then the result is a set containing only an empty sequence, or otherwise, the first observation is recorded in the merged trace.
  case-1 $1^\circ\ \ \langle\rangle \parallel_{X_1} \langle\rangle = \{\langle\rangle\}$

  $$2^\circ\ \ \langle (s_1, h) \rangle \frown t \parallel_{X_1} \langle\rangle = \begin{cases} \{\langle\rangle\} & \text{if } h \in X_1 \\ \{\langle (s_1, h) \rangle \frown l \mid l \in t \parallel_{X_1} \langle\rangle\} & \text{otherwise} \end{cases}$$

  where $h \in \{a, ch?v, ch!v, ch.v, s_1', \bot\}$
- When a communication is over a synchronous channel, if the first observations of two input traces match (see Definition 4 below), then a synchronisation may occur (denoted by the set $\mathcal{G}_1$) or at this moment

a synchronisation does not occur (denoted by the set $\mathcal{G}_2$). Otherwise, a synchronisation cannot occur. Here, two observations are matched provided that both channel input and output from two processes respectively are enabled under the same pre-state.

Definition 4 (Match). Given two pairs $p_1 = (s_1, h_1)$ and $p_2 = (s_2, h_2)$, where $h_1 \in \{ch?v_1, ch!v_1, ch.v_1\}$, $h_2 \in \{ch?v_2, ch!v_2, ch.v_2\}$, we say that they are matched if $s_1 = s_2$, $\{h_1, h_2\} = \{ch?v_1, ch!v_1\}$ and $v_1 = v_2$ are satisfied, denoted as $\mathsf{match}(p_1, p_2)$.

case-2 $\langle (s_1, h_1) \rangle \frown t_1 \parallel_{X_1} \langle (s_2, h_2) \rangle \frown t_2 = \begin{cases} \mathcal{G}_1 \cup \mathcal{G}_2 & \textit{if } \mathsf{match}((s_1, h_1), (s_2, h_2)) \\ \mathcal{G}_2 & \textit{otherwise} \end{cases}$

where $h_1 \in \{ch?v_1, ch!v_1, ch.v_1\}$, $h_2 \in \{ch?v_2, ch!v_2, ch.v_2\}$, $\mathcal{G}_1 \mathrel{\widehat{=}} \{\langle (s_1, ch.v) \rangle \frown l \mid l \in t_1 \parallel_{X_1} t_2\}$, and $\mathcal{G}_2 \mathrel{\widehat{=}} \{\langle (s_1, h_1) \rangle \frown l \mid l \in t_1 \parallel_{X_1} \langle (s_2, h_2) \rangle \frown t_2\} \cup \{\langle (s_2, h_2) \rangle \frown l \mid l \in \langle (s_1, h_1) \rangle \frown t_1 \parallel_{X_1} t_2\}$.

- When two actions (a_1 and a_2) are synchronised, there are four scenarios with respect to the initial states (s_1 and s_2) and actions from the first observations of two traces: (1) both actions are in the set X_1 but different or actions under different pre-states, (2) actions from X_1 are the same and under the same pre-state, (3) one of the actions is not in X_1, and (4) both actions are not in X_1. As shown in **case-3** below, the result is a set containing only an empty sequence for scenario (1). A synchronisation occurs under scenario (2), although it is postponed to occur under scenario (3). Either action can occur for scenario (4).

case-3 $\langle (s_1, a_1) \rangle \frown t_1 \parallel_{X_1} \langle (s_2, a_2) \rangle \frown t_2 =$

$$\begin{cases} \{\langle\rangle\} & a_1, a_2 \in X_1 \wedge (a_1 \neq a_2 \vee s_1 \neq s_2) \\ \{\langle (s_1, a_1) \rangle \frown l \mid l \in t_1 \parallel_{X_1} t_2\} & a_1, a_2 \in X_1 \wedge a_1 = a_2 \wedge s_1 = s_2 \\ \{\langle (s_2, a_2) \rangle \frown l \mid l \in \langle (s_1, a_1) \rangle \frown t_1 \parallel_{X_1} t_2\} & a_1 \in X_1 \wedge a_2 \notin X_1 \\ \{\langle (s_1, a_1) \rangle \frown l \mid l \in t_1 \parallel_{X_1} \langle (s_2, a_2) \rangle \frown t_2\} & \\ \cup & a_1 \notin X_1 \wedge a_2 \notin X_1 \\ \{\langle (s_2, a_2) \rangle \frown l \mid l \in \langle (s_1, a_1) \rangle \frown t_1 \parallel_{X_1} t_2\} & \end{cases}$$

- When the first observation of one input trace is in a waiting state (captured by $\perp$) which indicates the evaluation of the sequential program does not terminate, the result depends on the other first observation, (1) if both observations are in waiting states, the result is a set of either observation, (2) if the other first observation is an action requiring the synchronisation ($h \in X_1$), the result contains the waiting observation only, (3) or otherwise, either observation from two processes occurs. Note that a process could not perform any further if its trace ends with a state pair $(s, \perp)$.

case-4 $1^\circ \;\; \langle (s_1, \perp) \rangle \parallel_{X_1} \langle (s_2, \perp) \rangle = \{\langle (s_1, \perp) \rangle, \langle (s_2, \perp) \rangle\}$

$2^\circ \;\; \langle (s_1, \perp) \rangle \parallel_{X_1} \langle (s_2, h) \rangle \frown t = \begin{cases} \{\langle (s_1, \perp) \rangle\} & \textit{if } h \in X_1 \\ \{\langle (s_1, \perp) \rangle\} \cup \{\langle (s_2, h) \rangle \frown l \mid l \in \langle (s_1, \perp) \rangle \parallel_{X_1} t\} & \textit{otherwise} \end{cases}$

where $h \in \{a, ch?v, ch!v, ch.v, s_2'\}$

- When the first observation of one trace is an action a or a post-state s_1', and the other is a channel input $ch?v$, output $ch!v$, communication $ch.v$, or a post-state s_2', the merged observation is different, (1) if a is from the set X_1, then its occurrence is postponed ($\mathcal{G}_3$), (2) or otherwise, either observation from two processes occurs ($\mathcal{G}_3 \cup \mathcal{G}_4$).

case-5 $\langle (s_1, h_1) \rangle \frown t_1 \parallel_{X_1} \langle (s_2, h_2) \rangle \frown t_2 = \begin{cases} \mathcal{G}_3 & \textit{if } h_1 \in X_1 \\ \mathcal{G}_3 \cup \mathcal{G}_4 & \textit{otherwise} \end{cases}$

where $h_1 \in \{a, s_1'\}$, $h_2 \in \{ch?v, ch!v, ch.v, s_2'\}$, $\mathcal{G}_3 \mathrel{\widehat{=}} \{\langle (s_2, h_2) \rangle \frown l \mid l \in \langle (s_1, h_1) \rangle \frown t_1 \parallel_{X_1} t_2\}$, and $\mathcal{G}_4 \mathrel{\widehat{=}} \{\langle (s_1, h_1) \rangle \frown l \mid l \in t_1 \parallel_{X_1} \langle (s_2, h_2) \rangle \frown t_2\}$.

3.3.9. Interleave

In the open environment, processes P and Q run independently (except communications through synchronous channels) for $P \mid\mid\mid_{X_2} Q$. The semantics of the interleave operator defined below is based on the definition of parallel operator where the set X_1 is empty.

$P \mid\mid\mid_{X_2} Q \mathrel{\widehat{=}} P \parallel_{(\varnothing, X_2)} Q$

3.3.10. Hiding

The hiding operator makes all occurrences of actions in X_1 hidden from the environment of the process. Thus the actions in set X_1 are not recorded in the process trace.

$$P \setminus X_1 \mathrel{\widehat{=}} \mathbf{H}(\exists\, t \bullet P[t, X_1 \cup ref'/tr', ref'] \wedge (tr' - tr) = hide(t - tr, X_1)) \;;\; Skip$$

The definition of hiding is defined by renaming the final trace of P as t, and restricting t to the trace which contains all the events of process P except those in set X_1, captured by the function *hide*. The refusal set of P is the union of the final refusal set and set X_1. Note that Act denotes a set of actions.

$$hide : seq((\mathsf{S} \times \mathsf{E}) \cup (\mathsf{S} \times \mathsf{S}^{\perp})) \times \mathbb{P}\, Act \to seq((\mathsf{S} \times \mathsf{E}) \cup (\mathsf{S} \times \mathsf{S}^{\perp}))$$
$$hide(\langle\,\rangle, X_1) \mathrel{\widehat{=}} \langle\,\rangle$$
$$hide(\langle (s,e)\rangle \frown t, X_1) \mathrel{\widehat{=}} \begin{cases} hide(t, X_1) & \text{if } e \in X_1 \\ \langle (s,e)\rangle \frown hide(t, X_1) & \text{otherwise} \end{cases}$$

3.3.11. Refinement

Refinement calculus is designed to produce correct programs, assisting in the software development. In the UTP theory, it is expressed as logic implication; an implementation (denoted as predicate P) satisfying a specification (denoted as predicate S) is formally expressed by universal quantification implication $\forall\, a, a', \cdots \bullet P \Rightarrow S$, where $a, a', \cdots$ are all observational variables of the alphabet, which must be the same for the specification and implementation. The universal quantification implication is usually denoted as $[P \Rightarrow S]$. The definition of refinement in CSP# is given as below.

Definition 5 (Refinement). Let P and Q be predicates for CSP# processes with the same shared variable state space, the refinement $P \sqsupseteq Q$ holds iff $[P \Rightarrow Q]$.

The refinement ordering in our definition is strong; every observation that satisfies P must also satisfy Q. The observation includes all process behaviours, i.e., stability, termination, traces, and refusals. Moreover, the record of the trace considers both variable states and event occurrences. For example, given a process $P = [x = 2]b \to Skip \;\Box\; [x\ != 2]c \to Skip$, and a process $Q = [x = 2]b \to Skip \;\Box\; [x\ != 2]d \to Skip$, the refinement $P \sqsupseteq Q$ does not hold although one observation satisfies both processes when x is equal to 2. A counterexample is that when x is not equal to 2, processes P and Q perform action c and d, respectively.

Notice that we only allow that in the trace sequence of process P, every element shall be the same as its counterpart in Q. In other words, our refinement prevents atomic program operations updating shared variables from being refined by non-atomic program operations which make the same effect. For example, given a process $P = \{x = x + 1\} \to \{x = x + 1\} \to Skip$, and a process $Q = \{x = x + 2\} \to Skip$, the refinement $P \sqsupseteq Q$ does not hold.

Definition 6 (Equivalence). For any two CSP# processes P and Q, P is equivalent to Q if and only if $P \sqsupseteq Q \wedge Q \sqsupseteq P$.

3.3.12. Recursion

Let p be a variable standing for a call to a recursive process, F be a monotonic function from CSP# processes to CSP# processes, we consider the explicit definition $(\mu\, p \bullet F(p))$ of recursion whose semantics is defined as the weakest fixed point, which is the greatest lower bound of all the fixed points of P with the bottom element $H(true)$ and the top element $H(false)$; namely, $\sqcap\{p \mid p \sqsupseteq F(p)\}$.

3.3.13. Discussion

So far, we have defined the denotational semantics of CSP#. Based on the semantics, CSP# satisfies the following two properties.

Lemma 1 (Monotonicity). All process combinators defined in the CSP# language are monotonic[2].

[2] The proofs of Lemma 1 are available online at `http://www.comp.nus.edu.sg/~pat/semantics.zip`.

Theorem 3.1 (Compositionality). The open semantics of CSP# is compositional.

Proof. Given a process combinator F and processes P, Q such that P and Q are equivalent with respect to the open semantics, we have $P \sqsupseteq Q$ and $Q \sqsupseteq P$ according to Definition 6. According to Lemma 1, both $F(P) \sqsupseteq F(Q)$ and $F(Q) \sqsupseteq F(P)$ are valid, which indicates $F(P) = F(Q)$, i.e., the open semantics is compositional. □

4. Algebraic Laws

In this section, we present a set of algebraic laws concerning the distinct features of CSP#.

State Guard
guard - 1 $[b_1]([b_2]P) = [b_1 \wedge b_2]P$
guard - 2 $[b](P_1 \ \mathsf{op}\ P_2) = [b]P_1 \ \mathsf{op}\ [b]P_2$ where $\mathsf{op} \in \{\|, \Box, \sqcap\}$
guard - 3 $[false]P = Stop$
guard - 1 enables the elimination of nested guards. **guard - 2** shows the distribution of the state guard through parallel composition, external choice and internal choice. **guard - 3** shows that process $[false]P$ behaves like *Stop* because its guard can never be fired.

Sequential Composition
seq - 1 $(P_1 \ ; \ P_2) \ ; \ P_3 = P_1 \ ; \ (P_2 \ ; \ P_3)$
seq - 2 $P_1 \ ; \ (P_2 \sqcap P_3) = (P_1 \ ; \ P_2) \sqcap (P_1 \ ; \ P_3)$
seq - 3 $(P_1 \sqcap P_2) \ ; \ P_3 = (P_1 \ ; \ P_3) \sqcap (P_2 \ ; \ P_3)$
seq - 4 $P = Skip \ ; \ P$
seq - 5 $P = P \ ; \ Skip$
seq - 1 shows that sequential composition is associative. **seq - 2, 3** show the distribution of sequential composition through internal choice. **seq - 4, 5** show that process *Skip* is the left and right unit of sequential composition respectively, which show that CSP# processes also satisfy **CSP3** and **CSP4** healthiness conditions of CSP [HH98], i.e., $\mathbf{CSP3}(P) = Skip \ ; \ P$ and $\mathbf{CSP4}(P) = P \ ; \ Skip$.

Data Operation Prefixing
dat - 1 $\{\mathsf{while}\ true\ \mathsf{do}\ prog\} \to P = \{\mathsf{while}\ true\ \mathsf{do}\ prog\} \to Q$
dat - 2 $\{prog_1\} \to P \parallel_{(X_1,X_2)} \{prog_2\} \to Q = \{prog_1\} \to (P \parallel_{(X_1,X_2)} \{prog_2\} \to Q) \ \mathsf{op}\ \{prog_2\} \to (\{prog_1\} \to P \parallel_{(X_1,X_2)} Q)$ where $\mathsf{op} \in \{\Box, \sqcap\}$
dat - 3 $\{prog_1\} \to P \parallel_{(X_1,X_2)} h \to Q = \{prog_1\} \to (P \parallel_{(X_1,X_2)} h \to Q) \ \Box \ h \to (\{prog_1\} \to P \parallel_{(X_1,X_2')} Q)$
where $h \in \{ch?x, ch!exp, ch.v\}$, and X_2' is the set of all synchronous channel outputs and inputs of processes P and Q.

dat - 4 $\{prog_1\} \to P \parallel_{(X_1,X_2)} a \to Q = \begin{cases} \{prog_1\} \to (P \parallel_{(X_1,X_2)} a \to Q) & \text{if } a \in X_1 \\ \{prog_1\} \to (P \parallel_{(X_1,X_2)} a \to Q) \ \Box & \\ a \to (\{prog_1\} \to P \parallel_{(X_1,X_2)} Q) & \text{if } a \notin X_1 \end{cases}$
dat - 1 indicates that the loop operation would hold the execution of process. **dat - 2** indicates that data operations running in parallel do not need to be synchronised and are executed independently. **dat - 3** shows that the data operation and any type of channel communication running in parallel are executed independently. **dat - 4** shows the data operation should be executed firstly if a is the common action of process P and Q; otherwise, they are executed independently.

The following non-law shows that the data operation in CSP# is atomic; that is the program $prog_1; \ prog_2$ completes in a single observation.
nla - 1 $\{prog_1\} \to \{prog_2\} \to P \neq \{prog_1; \ prog_2\} \to P$

Parallel Composition
par - 1 $P_1 \parallel_{(X_1,X_2)} P_2 = P_2 \parallel_{(X_1,X_2)} P_1$
par - 2 $(P_1 \parallel_{(X_1,X_2)} P_2) \parallel_{(X_1,X_2)} P_3 = P_1 \parallel_{(X_1,X_2)} (P_2 \parallel_{(X_1,X_2)} P_3)$

par - 3 $Skip \parallel_{(\varnothing,X_2)} P = P = P \parallel_{(\varnothing,X_2)} Skip$ where X_2 is the set of all synchronous channel outputs and inputs of process P.
par - 1, 2 show that parallel composition is commutative and associative. Consequently, the order of parallel composition is irrelevant. **par - 3** shows that process *Skip* is the unit of parallelism.

Interleave
inter - 1 $P_1 \,|||_{X_2}\, P_2 = P_2 \,|||_{X_2}\, P_1$
inter - 2 $(P_1 \,|||_{X_2}\, P_2) \,|||_{X_2}\, P_3 = P_1 \,|||_{X_2}\, (P_2 \,|||_{X_2}\, P_3)$
inter - 3 $Skip \,|||_{X_2}\, P = P = P \,|||_{X_2}\, Skip$ where X_2 is the set of all synchronous channel outputs and inputs of process P.
inter - 1, 2 show that interleaving is commutative and associative. Consequently, the order of interleaving is irrelevant. **inter - 3** shows that process *Skip* is the unit of interleaving which shows that CSP# processes satisfy **CSP5** healthiness condition, i.e., $\mathbf{CSP5}(P) = P \,|||_{X_2}\, Skip$.

Hiding
hid - 1 $(P\backslash X)\backslash Y = (P\backslash Y)\backslash X$
hid - 2 $(P\backslash X)\backslash Y = P\backslash(X \cup Y)$
hid - 3 $P\backslash\varnothing = P$
hid - 4 $Skip\backslash X = Skip$
hid - 5 $(a \rightarrow P)\backslash X = \begin{cases} P\backslash X & \text{if } a \in X \\ a \rightarrow (P\backslash X) & \text{if } a \notin X \end{cases}$
hid - 6 $(\mu\, N \bullet (a \rightarrow N)) \setminus \{a\} = CHAOS$ where $CHAOS \mathrel{\widehat{=}} H(true)$
hid - 1 shows that the hiding operator is commutative and the order is not critical. **hid - 2** shows that the nested hiding can be combined. **hid - 3** shows that the hiding is void if the hidden set is empty. **hid - 4** shows that nothing will be hidden for the process *Skip*. **hid - 5** indicates that the behaviours of the hiding operation depends on the relation between a and X. **hid - 6** shows that if the only action in a recursive process is hidden, the behaviour of the hidden process leads to *CHAOS* is the worst CSP# process.

All algebraic laws can be established based on our denotational model. That is to say, if the equality of two syntactically different processes is algebraically provable, then the two processes are also equivalent with respect to the denotational semantics. Moreover, these algebraic laws can be used as auxiliary reasoning rules to prove process equivalence during theorem proving. Below we use **guard - 3** as an example to show that this particular law can be proved using our proposed denotational semantics. The proofs of the soundness of some other algebraic laws with respect to the denotational semantics are available online[3].
Law guard - 3 $[false]P = Stop$
Proof:

$$
\begin{array}{lll}
 & [false]P & [3.3.7] \\
= & \widehat{P} \lhd (\mathcal{B}(false)(\pi_1(head(tr' - tr))) = true \wedge tr < tr') \rhd Stop & [Def.\ 2] \\
= & \widehat{P} \lhd false \rhd Stop & [predicate\ calculus] \\
= & Stop & \square \\
where & \widehat{P} \mathrel{\widehat{=}} P \wedge tr < tr' \vee P(tr, tr) \wedge \exists\, s \in \mathsf{S} \cdot tr' - tr = \langle (s, s) \rangle &
\end{array}
$$

5. Encoding CSP# Denotational Semantics in PVS

So far, we have defined the denotational semantics of CSP#. To validate the consistency of our proposed semantics and further provide a machine-assisted support for verification, we encode the UTP semantics for CSP# in the PVS theorem prover [OSRSC01][4].

Our encoding includes three parts which are illustrated in the following subsections. First the theory of semantic model defines observational variables and healthiness conditions. Based on the semantic model theory, we define the theory of expressions and programs which encodes carefully the syntax and semantics of arithmetic expressions, Boolean expressions and sequential programs. Further, the semantics of processes

[3] `http://www.comp.nus.edu.sg/~pat/semantics.zip`
[4] PVS theories and proofs for CSP# semantics are available online at `http://www.comp.nus.edu.sg/~pat/semantics.zip`.

and refinement relationship are formalised in the theory of process. At the end of this section, we discuss machine-assisted proofs of important algebraic laws and lemmas based on the encoding in PVS.

5.1. The Theory of Semantic Model

The first challenge of the semantic model encoding is to develop an appropriate data structure to represent the observational variables and relations over observational variables. The second challenge is to capture different types of events dedicated to CSP#. To address these challenges, we adopt PVS abstract datatype constructor to handle event types and PVS set theory to model relations as illustrated below.

5.1.1. The Theory of Observational Variables

CSP# supports concurrency over communications and shared variables. We first define the shared variable state and event type in PVS. The following shows the formalised type for variable states (`S`) and all states (`S_abort`).

```
Vars: TYPE+
S: TYPE+ = [Vars -> int]
S_abort:  DATATYPE
 BEGIN
  abort: abort?
  is_S(left_s: S): is_S?
 END S_abort
```

In the above specification, type `S` is encoded as a function from variable type `Vars` to `int`. We define all states as a disjoint union by using a PVS abstract datatype: `abort` and `is_S` are constructors, predicates `abort?` and `is_S?` are recognisers of the type `[S_abort -> bool]`, which determine whether the argument is constructed using the corresponding constructor. Note that a similar fashion of applying the PVS abstract datatype is used throughout the rest of the section to model complex CSP# types.

CSP# supports both event synchronisation and pairwise handshake through synchronous channels. Thus event `E` includes actions, synchronous channel inputs, outputs and communications. To represent `E`, we define a datatype as follows.

```
E: DATATYPE WITH SUBTYPES RefE, Channelcom
 BEGIN
  action(ac:Ta): action?: RefE
  input(ci:Ti): input?: RefE
  output(co:To): output?: RefE
  chancom(cm:Tm): chancom?: Channelcom
 END E
```

Here, `Ta` is the type of actions, `Ti` is the type of channel inputs, `To` is the type of channel outputs, and `Tm` is the type of channel communications. Subtypes `RefE` and `Channelcom` denote the set of refused events and channel communications, respectively.

In our semantics, CSP# processes are interpreted as relations between initial observations and subsequent observations of their execution behaviours. Namely, relations are represented as predicates over observational variables. In Section 3.1.1, we have defined eight variables for the alphabet of CSP# semantics to capture all aspects of process behaviours. In PVS, we use a record type `AB` to represent this alphabet of the observations of CSP# process behaviours, and a set of such records to represent a relation.

```
AB: TYPE = [# ok:bool, ok1:bool, wait:bool, wait1:bool, ref:set[RefE], ref1:set[RefE],
              tr:Trace, tr1:Trace #]
Relation: TYPE = set[AB]
```

In the above formalisation, a dashed variable is represented by its undashed variable name suffixed with number 1, e.g., `ok1` denotes variable ok'. In our semantics for the trace, we use a sequence to record the observations on the interaction of the process with its environment. We use the PVS predefined datatype `list` below to represent the sequence. Thus an empty sequence can be represented by a null list (`null`), and sequence concatenation can be formalised by the predefined function `append` over lists. To simplify the PVS encoding, we define a function `snoc` for appending a single element to a list of type `SE` (the trace) and function `<=` for checking the trace prefixing relationship.

Predicate	PVS
$\neg P$	{pre:AB|NOT P(pre)}
$P \wedge Q$	{pre:AB|P(pre) AND Q(pre)}
$P \vee Q$	{pre:AB|P(pre) OR Q(pre)}
$P \Rightarrow Q$	{pre:AB|P(pre) IMPLIES Q(pre)}

Table 1. Predicate formalisation in PVS

```
S_E: DATATYPE
  BEGIN
    state(s1:S_abort): state?
    event(e:E): event?
  END S_E
SE: TYPE+ = [S, S_E]

t, t1: VAR Trace
se: VAR SE
snoc(t, se): Trace = append(t, cons(se, null))
<=(t,t1): bool = EXISTS (t2: Trace): t1 = append(t, t2)
```

Based on the formalisation of the alphabet and relation, we next illustrate how to formalise the detailed predicates in PVS. In general, a predicate P on the alphabet is encoded as a set {pre:AB|P(pre)}, and logic operators $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ are formalised as NOT, AND, OR, and IMPLIES respectively in PVS. A summary of the formalisation is shown in Table 1, where P and Q are predicates, and P and Q are sets in PVS.

5.1.2. The Theory of Healthiness Conditions

CSP# satisfies the healthiness conditions **R1** to **R3** for reactive processes.

```
P: VAR Relation
R1(P): Relation = {pre:AB|P(pre) AND pre‘tr <= pre‘tr1}
R2(P): Relation = {pre:AB|EXISTS(s:Trace): (EXISTS(pre0:AB): P(pre0) AND
  pre‘tr = s AND pre‘tr1 = append(s, pre0‘tr1-pre0‘tr) AND pre0‘tr <= pre0‘tr1 AND
  pre‘ok = pre0‘ok AND pre‘wait = pre0‘wait AND pre‘ref = pre0‘ref AND
  pre‘ok1 = pre0‘ok1 AND pre‘wait1 = pre0‘wait1 AND pre‘ref1 = pre0‘ref1)}
II: Relation = {pre:AB|(NOT pre‘ok AND pre‘tr <= pre‘tr1) OR (pre‘ok1 AND
  pre‘tr1 = pre‘tr AND pre‘wait1 = pre‘wait AND pre‘ref1 = pre‘ref)}
R3(P): Relation = {pre:AB|IF pre‘wait THEN II(pre) ELSE P(pre) ENDIF}
```

In our encoding, P is declared as a variable with type Relation, specified by P: VAR Relation. Healthiness condition **R1** is formalised as a function R1. Specifically R1 takes an arbitrary relation P as an input and returns a relation satisfying a predicate which is modelled as as a set of records; each record is a member of the relation (denoted as P(pre)) and its final trace extends the initial trace (denoted as pre‘tr <= pre‘tr1). Function R2 specifies that observation of the process is not changed given the value s of pre‘tr made arbitrary. Function R3 specifies that for each element pre, if the value of pre‘wait is true, then pre is a member of the reactive identity relation II, or otherwise, it is a member of relation P.

In addition, CSP# satisfies two healthiness conditions **CSP1** and **CSP2** for communicating sequential processes, defined as follows.

```
CSP1(P): Relation = {pre: AB|(NOT pre‘ok AND pre‘tr <= pre‘tr1) OR P(pre)}
CSP2(P): Relation = {pre: AB|EXISTS (p:AB): P(p) AND
  (pre‘ok = p‘ok AND pre‘wait = p‘wait AND pre‘ref = p‘ref AND pre‘tr = p‘tr) AND
  ((p‘ok1 => pre‘ok1) AND pre‘tr1 = p‘tr1 AND pre‘wait1 = p‘wait1 AND pre‘ref1 = p‘ref1)}
```

Function CSP1 denotes that for each element pre, when the value of pre‘ok is false, its trace shall be extended, or it remains unchanged. The sequential composition ; in $\mathbf{CSP2}(P) = P\ ;\ ((ok \Rightarrow ok') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref)$ is explicitly formalised; namely, for each element pre, its undashed variable value is the same as the undashed variable value of an element from relation P, e.g., pre‘ok = p‘ok, while its dashed variable value is the same as the dashed variable value from the second program, e.g., ok' is represented by pre‘ok1. Meanwhile, the dashed variable value of an element from relation P is the same as the undashed variable value in the second program, e.g., ok in the second program is represented by p‘ok1.

Finally, processes in CSP# are defined by satisfying all the healthiness conditions. Our definition of processes relies on PVS subtyping: process is a subtype of Relation, where function H is the composition of five healthiness condition functions, and predicate H(P) = P depicts relation P satisfies the condition H.

```
H(P): Relation = CSP2(CSP1(R3(R2(R1(P)))))
process: TYPE = {P| H(P) = P}
```

5.2. The Theories of Expressions and Programs

We present the encoding of program syntax and semantics as below. For simplicity, we omit the encoding of expressions here; the detailed encoding is available online at `http://www.comp.nus.edu.sg/~pat/semantics.zip`.

```
 1  % program syntax
 2  Prog: Datatype
 3   BEGIN
 4    skip: skip?
 5    abort_prog: abort_prog?
 6    assign(x:Vars, exp:Aexp): assign?
 7    seq(prog1,prog2: Prog): seq?
 8    if_prog(ifcond:Bexp, thn,els:Prog): if?
 9    while(whilecond:Bexp, body:Prog): while?
10   END Prog
11  % state relation
12  S_S_abort: TYPE+ = [S, S_abort]
13  Prog_State: TYPE = set[S_S_abort]
14  % type for arithmetic and Boolean expressions
15  S_int: TYPE+ = [S -> int]
16  S_bool: TYPE+ = [S -> bool]
17  % skip
18  skip_ceval: Prog_State = {state:S_S_abort| state`2 = is_S(state`1)}
19  %abort_prog
20  abort_ceval: Prog_State = {state:S_S_abort| true}
21  % update
22  update_ceval (x: Vars, v: S_int): Prog_State = {state:S_S_abort|
      state`2 = is_S(state`1 WITH [(x) := v(state`1)])}
23  % sequence
24  seq_ceval(s_prog1, s_prog2: Prog_State): Prog_State = {state:S_S_abort|
       IF s_prog1(state`1, abort) THEN state`2 = abort
       ELSE (EXISTS (mid:S): s_prog1(state`1, is_S(mid)) AND s_prog2(mid,state`2) ENDIF}
25  % if
26  if_ceval(s_b:S_bool, s_prog1, s_prog2: Prog_State): Prog_State =
      {state:S_S_abort|IF s_b(state`1) THEN s_prog1(state`1,state`2)
       ELSE s_prog2(state`1,state`2) ENDIF}
27  % while
28  while_ceval(s_b:S_bool, s_prog: Prog_State): Prog_State =
       mu (lambda (X:Prog_State): if_ceval(s_b, seq_ceval(s_prog, X), skip_ceval))
29  % sequential program semantics
30  ceval(p:Prog): RECURSIVE  Prog_State =
31    (CASES p of
32      skip: skip_ceval,
33      abort_prog: abort_ceval,
34      assign(x,exp): update_ceval (x, aeval(exp)),
35      seq(prog1,prog2): seq_ceval(ceval(prog1),ceval(prog2)),
36      if_prog(ifcond,thn,els): if_ceval(beval(ifcond),ceval(thn),ceval(els))
37      while(wcond,bprog): while_ceval(beval(wcond), ceval(bprog))
38     ENDCASES)
39  MEASURE p BY <<
```

In the above PVS specifications, we define data type `Prog` to encode the program syntax, data type `Aexp` for the syntax of arithmetic expressions, and data type `Bexp` for the syntax of Boolean expressions. We further model the program semantics by a recursive function `ceval`, which uses PVS case construct to capture different program types in `Prog`. For example, the evaluation of sequential composition program (at line 35) is defined by a function `seq_ceval` which returns a set of all possible state pairs `state`, if the evaluation of program `s_prog1` does not terminate under a pre-state `state`1` (captured by predicate `s_prog1(state`1, abort)` being true), then the post-state `state`2` is `abort` representing nontermination (at line 24), or otherwise, the post-state is the final state after the execution of two sequential programs (at line 24). Note that the type of `mid` is `S`, if it is directly encoded as the output of `s_prog1`, the typechecking

Operation	CSP#	PVS
Stop	*Stop*	`Stop`
Skip	*Skip*	`Skip`
event prefixing	$a \rightarrow P$	`a >> P`
channel output	$ch!exp \rightarrow P$	`ch_o_exp >> P`
channel input	$ch?m \rightarrow P(m)$	`ch_i_m >> Pi`
data operation prefixing	$\{prog\} \rightarrow P$	`prog >> P`
state guard	$[b]P$	`[||](b,P)`
external choice	$P \Box Q$	`P <> Q`
internal choice	$P \sqcap Q$	`P \/ Q`
sequential composition	$P\ ;\ Q$	`P ++ Q`
hiding	$P \setminus X_1$	`Hid(P,X1)`
parallel	$P \parallel_{(X_1,X_2)} Q$	`Par(P, Q)(X1, X2)`
interleaving	$P \parallel\parallel_{X_2} Q$	`Inter(P, Q)(X2)`

Table 2. CSP# process syntax

finds an error which is a wrong type of the second argument to `s_prog1`; namely, it expects type `S_abort` but finds type `S`, thus we use the constructor to cast the type to be `S_abort` (denoted as `is_S(mid)`). Here, recursive functions `aeval(a: Aexp): RECURSIVE S_int` at line 34 and `beval(b: Bexp): RECURSIVE S_bool` at line 36 evaluate arithmetic and Boolean expressions respectively. We encode the semantics of the while-loop program at line 37 following the existing effort in [PDvHR96]. The recursive function `ceval` must terminate, which is specified by the measure expression at line 39. The expression follows the `MEASURE` keyword and ends with an order relation `<<` which specifies an irreflexive subterm function/predicate. PVS generates four termination type correctness conditions (TCCs) and we have discharged the generated proof obligations.

5.3. The Theory of Processes

PVS has a fixed syntax, and users cannot introduce new symbols. Thus we cannot directly use the standard CSP# process notations. Instead, we use the existing symbols in PVS, and summarize the standard CSP# syntax (Section 2.1) and our PVS encoding in Table 2, where `X1` is a set of actions of the type `Ta`, and process `Pi` is a parametric process of the type `[int -> process]`. Note that recursive processes cannot be directly defined in PVS. We will apply `mucalculus` theory in PVS to formalising them, and the detailed formalisation is presented in Section 5.3.12.

5.3.1. Primitives

Following the guideline on how to formalise a relation in PVS (Section 5.1.1), it is straightforward to define the primitive processes *Stop* and *Skip* in PVS, shown as follows. Here, `pre` is a declared as a variable with type `AB`.

```
pre: VAR AB
Stop: process = H({pre|pre`ok1 AND pre`tr1=pre`tr AND pre`wait1});
Skip: process = H({pre|(NOT pre`ok AND pre`tr<=pre`tr1) OR
  (pre`ok1 AND pre`tr1=pre`tr AND pre`wait1=pre`wait)});
```

5.3.2. Sequential Composition

The semantics of sequential composition $P\ ;\ Q$ is defined as the merge of two processes with the value of dashed observational variables in P being the same as the value of undashed variables in Q. The formalisation of sequential composition in PVS is defined as follows.

```
p, q: VAR AB
P, Q: VAR process
equateUndashed(p,q): bool = p`ok=q`ok AND p`wait=q`wait AND p`ref=q`ref AND p`tr=q`tr;
equateMiddle(p,q): bool = p`ok1=q`ok AND p`wait1=q`wait AND p`ref1=q`ref AND p`tr1=q`tr;
equateDashed(p,q): bool = p`ok1=q`ok1 AND p`wait1=q`wait1 AND p`ref1=q`ref1 AND p`tr1=q`tr1;
++(P, Q): process = {pre|EXISTS (p,q: AB): P(p) AND Q(q) AND
  equateUndashed(p, pre) AND equateMiddle(p,q) AND equateDashed(q,pre)};
```

We define three auxiliary functions to capture the merging operations. Function `equateUndashed (p, q)` equates the undashed observational variable values of `p` to `q`'s, function `equateMiddle(p,q)` equates the dashed observational variable values of `p` to the undashed ones of `q`, and function `equateDashed(p,q)` equates the dashed observational variable values of `p` to `q`'s. In our encoding for sequential composition, for each element `pre`, its undashed variable values are the same as the undashed ones of an element `p` from process `P` (captured by `equateUndashed(p, pre)`), and its dashed variable values are the same as the dashed ones of an element `q` from `Q` (captured by `equateDashed(q,pre)`). Meanwhile, the dashed variable values of an element `p` from `P` is the same as the undashed ones of an element `q` from `Q` (captured by `equateMiddle(p,q)`).

5.3.3. Event prefixing

The formalisation of event prefixing `a >> P` contains two parts: first we define a prefixed action `Skip(a)`, and then compose the action with process `P`. Variable `a` is an action with type `Ta`.

```
Ta: TYPE
a: VAR Ta
Skip(a): process = H({pre|pre‘ok1 AND
  ((pre‘wait1 AND NOT pre‘ref1(action(a)) AND pre‘tr1=pre‘tr) OR
   (NOT pre‘wait1 AND EXISTS(s:S): (pre‘tr1=snoc(pre‘tr, (s,event(action(a))))))))});
>>(a, P): process = (Skip(a) ++ P);
```

Here our predefined function `snoc` appends a pair `(s,event(action(a)))` consisting of a pre-state and an event to the end of the list `pre‘tr`.

5.3.4. Synchronous Channel Output/Input

We first construct three tuples to respectively represent the type of synchronous channel input, output, and communication. Each tuple consists of three elements: the first is a string denoting a channel name, the second is a flag denoting the communication type, and the third is a number indicating the message.

```
% Type for input, output and communication symbol
T_i: TYPE+ = i
T_o: TYPE+ = o
T_m: TYPE+ = m
% Type for channel input, output and communication
Ti: TYPE = [string, T_i, int]
To: TYPE = [string, T_o, int]
Tm: TYPE = [string, T_m, int]
```

We next encode the syntax of channel input and output into PVS, similar to the way of defining above, where type `Vars` denotes variable names (Section 5.1.1) and type `Aexp` denotes the syntax of arithmetic expressions (Section 5.2).

```
% Syntax type for channel output/input/communication
Ti_syntax: TYPE = [string, T_i, Vars]
To_syntax: TYPE = [string, T_o, Aexp]
```

Based on the above encoding of event type and syntax definitions, the synchronous channel output is defined as follows.

```
1  ch_o_exp: VAR To_syntax %("ch",o,exp)
2  OutC(ch_o_exp): process = H({pre|pre‘ok1 AND
3    ((pre‘wait1 AND FORALL (v:int):(NOT pre‘ref1(input((ch_o_exp‘1,i,v)))) AND pre‘tr1=pre‘tr) OR
4     (NOT pre‘wait1 AND EXISTS(s:S): (pre‘tr1=snoc(pre‘tr,
       (s,event(output((ch_o_exp‘1,ch_o_exp‘2,aeval(ch_o_exp‘3)(s))))))))});
5  >>(ch_o_exp, P): process = (OutC(ch_o_exp) ++ P);
```

In our proposed denotational semantics, predicate $ch? \notin ref'$ denotes a process refuses all inputs. Here, we constrain explicitly that no input is in the refused set (shown at line 3).

Different from the above encoding of synchronous channel output, the formalisation of synchronous channel input below takes parametric process `Pi` into account. We model the input prefixing by a set of observational variable records, where each record is a member of the sequential composition of a channel input process `InC(ch_i_m, v)` and process `Pi(v)`. Value `v` denotes a possible message. In this way, parametric process `Pi` can also be applied to multiple indexed processes, for example, process `Pi` can be of the type `[int,...,int -> process]`.

```
ch_i_m: VAR Ti_syntax %("ch",i,m)
InC(ch_i_m, v): process = H({pre|pre`ok1 AND
  ((pre`wait1 AND FORALL (v1:int):(NOT pre`ref1(output((ch_i_m`1,o,v1)))) AND pre`tr1=pre`tr) OR
   (NOT pre`wait1 AND EXISTS(s:S):(pre`tr1=snoc(pre`tr, (s,event(input((ch_i_m`1,ch_i_m`2,v))))))))});
>>(ch_i_m, Pi): process = {pre|EXISTS(v:int):member(pre,(InC(ch_i_m, v) ++ Pi(v)))};
```

5.3.5. Data Operation Prefixing

The formalisation of data operation prefixing process `prog >> P` contains two parts: first we define a data operation process `Data(prog)`, and then compose the data operation with process `P`. In our formalisation, `prog` is declared as a variable with the sequential program type `Prog`, `ceval` is a function for evaluating the program (formalised in Section 5.2), and `ceval(prog)(s,abort)` is a predicate capturing that the evaluation of the program does not terminate (represented by predicate $(s, \perp) \in \mathcal{C}[\![prog]\!]$ in the denotational semantics of data operation prefixing).

```
prog: VAR Prog
Data(prog): process = H({pre|pre`ok1 AND EXISTS(s:S):
  ((pre`wait1 AND ceval(prog)(s,abort) AND
    pre`tr1=snoc(pre`tr,(s,state(abort)))) OR
   (NOT pre`wait1 AND NOT ceval(prog)(s,abort) AND
    EXISTS(s1:S):(ceval(prog)(s,is_S(s1)) AND pre`tr1=snoc(pre`tr,(s,state(is_S(s1))))))
  )});
>>(prog, P): process = (Data(prog) ++ P);
```

5.3.6. Choice

The internal choice indicates that process $P \sqcap Q$ behaves like either P or Q.

```
\/(P, Q): process = {pre|P(pre) OR Q(pre)};
```

Regarding the external choice of two processes P and Q, if no observation has been made and termination has not occurred (i.e., indicated by `Stop(pre)`), then it behaves like the conjunction of P and Q, or otherwise, it behaves as the disjunction. The formalisation of external choice is shown below.

```
<>(P, Q): process = H({pre|(Stop(pre) AND P(pre) AND Q(pre)) OR
  (NOT Stop(pre) AND (P(pre) OR Q(pre)))});
```

5.3.7. State Guard

The behaviour of process $[b]P$ is determined by the evaluation of Boolean condition b. The evaluation is modelled by function `g_beval`, which first checks whether two input traces fulfill an extension relationship (specified by an overloading function `<(t,t1)` for trace prefixing at line 3). If no, `g_beval` returns false (at line 7). Otherwise, `g_beval` evaluates b (`beval(b)`) under the pre-state of the first element from the extended trace (specified by (nth((t1-t),0))`1 at line 6). In our encoding, the PVS predefined function `nth` returns the nth element from the trace. For example, `nth((t1-t),0)` represents the predicate $(head(t1 - t))$ describing the initial observation on the trace of a process.

Further we define a function `PSkip(P)` at line 9 to formalise the semantics of a variant process $\widehat{P}$ in Section 3.3.7 which adds a stuttering step. The state guard process is formalised as function `[||](b,P)` at line 10 which indicates that the process behaves as the variant process (represented by `PSkip(P)(pre)`) if the guard is satisfied under the pre-state of the initial observation (represented by `g_beval(b,pre`tr,pre`tr1)`), or otherwise it behaves as process *Stop* (represented by `Stop(pre)`).

```
1  b: VAR Bexp
2  t, t1: VAR Trace
3  <(t,t1): bool = EXISTS (t2: Trace):(t2/=null AND t1=append(t,t2));
4  g_beval(b, t, t1): bool =
5    IF <(t,t1) THEN
6      beval (b)((nth(t1-t,0))`1)
7    ELSE false
8    ENDIF;
9  PSkip(P): process = {pre|P(pre) AND pre`tr<pre`tr1 OR
     EXISTS (pre1:AB),(s:S):(pre1`tr1=pre1`tr AND P(pre1) AND
     pre = pre1 with [`tr1 :=snoc(pre1`tr,(s,state(is_S(s))))])};
```

```
10  [||](b,P): process = {pre|(g_beval(b,pre‘tr,pre‘tr1) AND pre‘tr<pre‘tr1 AND PSkip(P)(pre))
      OR (NOT (g_beval(b,pre‘tr,pre‘tr1) AND pre‘tr<pre‘tr1) AND Stop(pre))};
```

5.3.8. Parallel Composition

As mentioned in Section 3.3.8, the semantics of parallel composition captures different types of merge. We model parallel composition in PVS below. Set `X1` denotes the common actions of process P and Q, and set `X2` denotes all synchronous channel inputs and outputs of the two processes.

```
 1  P, Q: VAR process
 2  Tio: TYPE ={x:RefE|EXISTS(chi:Ti): x=input(chi) OR EXISTS(cho:To): x=output(cho)}
 3  X1: VAR set[Ta]
 4  X2: VAR set[Tio]
 5  Par(P, Q)(X1, X2): process = H({pre|EXISTS (p,q: AB, X1a: set[RefE]): P(p) AND Q(q) AND
 6    equateUndashed(pre,p) AND equateUndashed(pre,q) AND
 7    pre‘ok1=(p‘ok1 AND q‘ok1) AND pre‘wait1=(p‘wait1 OR q‘wait1) AND
 8    pre‘ref1= union(union(inter(inter(p‘ref1,q‘ref1),X2), inter(union(p‘ref1,q‘ref1), X1a)),
       (inter(p‘ref1,q‘ref1)-X1a-X2)) AND
 9    X1a={x:RefE|EXISTS(a: Ta): X1(a) AND x=action(a)} AND
10    member(pre‘tr1-pre‘tr, tr_syn(p‘tr1-p‘tr, q‘tr1-q‘tr, X1))})
```

In the above formalisation, the value of an undashed observational variable of the parallel composition is the same as the counterpart of processes P and Q, captured by auxiliary functions `equateUndashed(pre,p)` and `equateUndashed(pre,q)` defined in Section 5.3.2 at line 6. On the other hand, the dashed variables capture four kinds of behaviour: 1. the divergence (`pre‘ok1=(p‘ok1 AND q‘ok1)` at line 7), 2. the termination (`pre‘wait1=(p‘wait1 OR q‘wait1)` at line 7), 3. the refusal (at lines 8-9) where we construct a set `X1a` whose elements are of the type `RefE` from the set `X1`, 4. the trace of the composition is a member of the set of traces produced by the trace synchronisation function `tr_syn` at line 10.

We remark that the definition of the trace synchronisation function in Section 3.3.8 is composed of five cases. In PVS, we formalise the definition accordingly. The specification below at lines 4-9 and 12-13 covers the scenarios where one of the traces is an empty trace; we use `null` to indicate an empty trace. Line 5 models the result when both traces are empty (**case-1** 1°). When one of the trace is not empty, the result is determined by the first observation of the non-empty trace. For example, lines 7-8 divide the the first case (**case-1** 2°) into two branches depending on whether the observation is an action (checked by recognisers `event?` and `action?`) and the action is in the set `X1` (checked by `X1(ac(e(se1‘2)))`). Line 8 depicts the remaining scenarios where the first observation is not in the set `X1`, or it is a synchronous channel input/output/communication or a state transition. Lines 12-13 formalise the definition when the second trace is empty. Note that we have to formalise all scenarios of the definition of the trace synchronisation function although the definition in Section 3.3.8 is simplified by imposing the symmetric property.

The specification at lines 15-18 formalises **case-2**, i.e., communications are over synchronous channels (checked by the predicates in the if condition at line 15). If the match condition is valid (line 16), there are two possible behaviours: whether a synchronous channel communication occurs or not at this moment (line 17). Otherwise, line 18 handles the case when the condition is invalid. We define a function `chanCheck` to check whether the first observations of two traces are channel inputs/outputs/communications, a function `match` to check whether the synchronisation is between one synchronous channel input and a corresponding output, and a function `aux_chancom` to specify the synchronised channel communication. In particular, the `match(s1, e1, s2, e2)` function checks the equivalence of three entities: the pre-states of two input traces (`s1=s2`), the channel names (e.g., `(co(e1))‘1=(ci(e2))‘1`), and the messages through the channel (e.g., `(co(e1))‘3=(ci(e2))‘3`).

```
e1, e2: VAR E
s1, s2: VAR S
se1: VAR SE
chanCheck(e1, e2): bool = (output?(e1) OR input?(e1) OR chancom?(e1))
  AND (output?(e2) OR input?(e2) OR chancom?(e2))
match(s1, e1, s2, e2): bool = s1=s2 AND
  (output?(e1) AND input?(e2) AND (co(e1))‘1=(ci(e2))‘1 AND (co(e1))‘3=(ci(e2))‘3 OR
   input?(e1) AND output?(e2) AND (ci(e1))‘1=(co(e2))‘1 AND (ci(e1))‘3=(co(e2))‘3)
aux_chancom(se1): set[SE] = {se3:SE|EXISTS(chm:Tm): se3‘1=se1‘1 AND event?(se1‘2) AND
  event?(se3‘2) AND chancom?(e(se3‘2)) AND e(se3‘2)=chancom(chm) AND chm‘2=m AND
  (output?(e(se1‘2)) AND chm‘1=(co(e(se1‘2)))‘1 AND chm‘3=(co(e(se1‘2)))‘3 OR
   input?(e(se1‘2)) AND chm‘1=(ci(e(se1‘2)))‘1 AND chm‘3=(ci(e(se1‘2)))‘3)}
```

```
 1  tr1, tr2: VAR Trace
 2  tr_syn(tr1, tr2, X1): RECURSIVE set[Trace] =
 3    CASES tr1 OF
 4      null: CASES tr2 OF
 5              null: {t1: Trace|t1=null},
 6              cons(se,t2):
 7                 IF event?(se‘2) AND action?(e(se‘2)) AND X1(ac(e(se‘2))) THEN {t:Trace|t=null}
 8                 ELSE {t:Trace|EXISTS(l:Trace):t=cons(se,l) AND member(l, tr_syn(tr1,t2,X1))} ENDIF
 9            ENDCASES,
10      cons(se1, t1):
11        CASES tr2 OF
12          null: IF event?(se1‘2) AND action?(e(se1‘2)) AND X1(ac(e(se1‘2))) THEN {t1:Trace|t1=null}
13                ELSE {t:Trace|EXISTS(l:Trace): t=cons(se1,l) AND member(l, tr_syn(t1,tr2,X1))}  ENDIF,
14          cons(se2, t2):
15            IF event?(se1‘2) AND event?(se2‘2) AND chanCheck((e(se1‘2)), (e(se2‘2))) THEN
16              IF match(se1‘1, e(se1‘2), se2‘1, e(se2‘2)) THEN
17                {t3:Trace|(EXISTS(se3:SE)(l:Trace):member(se3,aux_chancom(se1)) AND t3=cons(se3,l) AND
                   member(l, tr_syn(t1,t2,X1))) OR (EXISTS(l: Trace): t3=cons(se1,l) AND
                   member(l,tr_syn(t1,tr2,X1)) OR t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X1)))}
18              ELSE {t3:Trace|EXISTS(l: Trace): t3=cons(se1,l) AND member(l,tr_syn(t1,tr2,X1)) OR
                     t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X1))}  ENDIF
19            ELSIF event?(se1‘2) AND event?(se2‘2) AND action?(e(se1‘2)) AND action?(e(se2‘2)) THEN
20              IF X1(ac(e(se1‘2))) AND X1(ac(e(se2‘2))) AND (se1‘1 /= se2‘1 OR se1‘2 /= se2‘2) THEN
21                {t1: Trace|t1=null}
22              ELSIF X1(ac(e(se1‘2))) AND X1(ac(e(se2‘2))) AND se1‘1=se2‘1 AND se1‘2=se2‘2 THEN
23                {t3:Trace|EXISTS(l:Trace): t3=cons(se1,l) AND member(l,tr_syn(t1,t2,X1))}
24              ELSIF X1(ac(e(se1‘2))) AND NOT X1(ac(e(se2‘2))) THEN
25                {t3:Trace|EXISTS(l:Trace): t3=cons(se2,l) AND member(l,tr_syn(tr1,t2,X1))}
26              ELSIF NOT X1(ac(e(se1‘2))) AND X1(ac(e(se2‘2))) THEN
27                {t3:Trace|EXISTS(l:Trace): t3=cons(se1,l) AND member(l,tr_syn(t1,tr2,X1))}
28              ELSE {t3:Trace|EXISTS(l:Trace): (t3=cons(se1,l) AND member(l,tr_syn(t1,tr2,X1))) OR
                     (t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X1)))}  ENDIF
29            ELSIF state?(se1‘2) AND abort?(s1(se1‘2)) THEN
30              IF state?(se2‘2) AND abort?(s1(se2‘2)) THEN {t3:Trace|t3=tr1 OR t3=tr2}
31              ELSIF event?(se2‘2) AND action?(e(se2‘2)) AND X1(ac(e(se2‘2))) THEN {t3:Trace|t3=tr1}
32              ELSE {t3:Trace|t3=tr1 OR EXISTS(l:Trace):t3=cons(se2,l) AND member(l,tr_syn(tr1,t2,X1))}
33              ENDIF
34            ELSIF state?(se2‘2) AND abort?(s1(se2‘2)) THEN
35              IF state?(se1‘2) AND abort?(s1(se1‘2)) THEN {t3:Trace|t3=tr1 OR t3=tr2}
36              ELSIF event?(se1‘2) AND action?(e(se1‘2)) AND X1(ac(e(se1‘2))) THEN {t3:Trace|t3=tr2}
37              ELSE {t3:Trace|t3=tr2 OR EXISTS(l:Trace):t3=cons(se1,l) AND member(l,tr_syn(t1,tr2,X1))}
38              ENDIF
39            ELSIF condCheck(se1,se2,X1) THEN
40              {t3:Trace|EXISTS(l:Trace): t3=cons(se2,l) AND member(l,tr_syn(tr1,t2,X1))}
41            ELSIF condCheck(se2,se1,X1) THEN
42              {t3:Trace|EXISTS(l:Trace): t3=cons(se1,l) AND member(l,tr_syn(t1,tr2,X1))}
43            ELSE {t3:Trace|EXISTS(l:Trace): (t3=cons(se1,l) AND member(l,tr_syn(t1,tr2,X1)))
                    OR (t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X1)))}
44            ENDIF
45        ENDCASES
46    ENDCASES
47    MEASURE length(tr1)+length(tr2)
```

When encoding **case-3**, namely, synchronisation between two actions (line 19), there are four scenarios: 1. both actions are in the set `X1` but different or from different pre-states (lines 20-21), 2. actions are the same and from the same pre-states (lines 22-23), 3. an action is not in `X1` (lines 24-27), and 4. both actions are not in `X1` (line 28).

The specification at lines 29-38 covers the scenarios (**case-4**) where the first observation is in a waiting state (e.g., `state?(se1‘2) AND abort?(s1(se1‘2))` at line 29). Line 30 models the result when the first observations of both traces are non-terminations (**case-4** 1°). When the first observation of the second trace is not in a waiting state, the result is determined by the first observation of the second trace (**case-4** 2°). If it is an action in `X1`, the result only contains the non-termination (line 31), or otherwise, either observation occurs (line 32). The formalisation of the case where the first observation of the second trace is in a waiting state is similar, specified at lines 34-38.

When formalising **case-5** that a synchronisation is between an action or a state transition and a synchronous channel input/output/communication or a state transition (specified by the auxiliary function `condCheck` at lines 39 and 41), two scenarios are considered: if one observation is an action in `X1`, then its occurrence is postponed (lines 39-42), or otherwise, either observation occurs (lines 43-44).

```
se2: VAR SE
condCheck(se1, se2, X1): bool = event?(se1‘2) AND action?(e(se1‘2)) AND X1(ac(e(se1‘2))) AND
  (event?(se2‘2) AND (input?(e(se2‘2)) OR output?(e(se2‘2)) OR chancom?(e(se2‘2)))
   OR state?(se2‘2) AND is_S?(s1(se2‘2)))
```

5.3.9. Interleave

The semantics of interleaving process $P \,|||\, Q$ is similar to the semantics of parallel operator, except the set X which only contains synchronous channel outputs and inputs.

```
Inter(P, Q)(X2): process = Par(P, Q)(emptyset,X2)
```

5.3.10. Hiding

In process $P \setminus X_1$, all the occurrence of actions in set X_1 are not observed or controlled by the environment. We formalise the hiding operator as follows.

```
1  X1: VAR set[Ta]
2  Hid(P,X1): process = H(pre|EXISTS (s:AB, Xa:set[RefE]): P(s) AND equateUndashed(pre,s) AND
3    pre‘ok1=s‘ok1 AND pre‘wait1=s‘wait1 AND pre‘tr1=snoc(pre‘tr, hide(s‘tr1-s‘tr,X1)) AND
4    s‘ref1=union(pre‘ref1, Xa) AND Xa={refa:RefE|EXISTS (a: Ta): X1(a) AND refa = action(a)}) ++ Skip;
```

The above definition depicts that the behaviour of hiding process is the same as `P` except the final value of refusals and traces. Specifically, the refusals are defined as that from `P` excluding the hiding actions in `X1`. Since the refusals has the type `set[RefE]` and set X1 is of the type `set[Ta]`, we cannot directly use the set extraction function in PVS. Instead, we construct a set `Xa` whose elements are of the type `RefE` and contains actions in set `X1` only (specified by line 4). Auxiliary function `equateUndashed(pre,s)` defined in Section 5.3.2 returns true if the values of undashed variables (i.e., `ok`, `wait`, `trace`, and `ref`) of records `pre` and `s` are the same. To model the actions hiding in a trace, we define a function `hide` which is a recursive function over the trace structure.

```
1  hide(t,X1): RECURSIVE Trace =
2    CASES t OF
3      null: null,
4      cons(x1, x2): (IF event?(x1‘2) AND action?(e(x1‘2)) AND  X1(ac(e(x1‘2))) THEN
5                       hide(x2,X1)
6                     ELSE  cons(x1,hide(x2,X1))
7                     ENDIF)
8  ENDCASES
9  MEASURE length(t)
```

Here, we use PVS `CASES` expressions to discuss two patterns of the trace. If the trace contains an action from set `X1` (specified as `event?(x1‘2) AND action?(e(x1‘2)) AND X1(ac(e(x1‘2)))` at line 4), then we remove this action from the trace (specified as `hide(x2,X1)` at line 5). Otherwise, this trace is unchanged (specified as `cons(x1,hide(x2,X1)) at line 6`).

5.3.11. Refinement

We define symbol `|>` to represent the refinement operator ($\sqsupseteq$) in the following PVS specifications, where the PVS predefined function `subset?` checks whether set `P` is a subset of set `Q`. Namely, process P refines Q iff the formalisation `P` is a subset of `Q` in our encoding.

```
|>(P,Q): bool = subset?(P,Q)
```

5.3.12. Recursion

In CSP#, the semantics of a recursive process is defined using the weakest fixed point, which is the greatest lower bound of all the fixed points. PVS provides the formalisation of fixed points for sets, but we cannot directly use it because the process in our calculus is defined as set of constrained object, i.e, healthy predicate.

Following the `mucalculus` theory in PVS, we define our own greatest lower bound `glb(SX)` of any set `SX` of processes, whereas `monotonic?(F)` which checks whether `F` is a monotonic mapping, and `mu(G)` which represents the weakest fixed point given a monotonic mapping `G`. Nonetheless, the main properties of weakest fixed point are still valid in our formalisation shown below.

```
SX: VAR set[process]
X, Y: VAR process
pre: VAR AB

glb(SX): process = H({pre|EXISTS (X: (SX)): X(pre)})
F: VAR [process -> process]
monotonic?(F): bool = FORALL X,Y: X |> Y IMPLIES F(X) |> F(Y)
G: VAR (monotonic?)
mu(G): process = glb({X|X |> G(X)})

closure_mu: LEMMA mu(G) |> G(mu(G))
smallest_closed: LEMMA X |> G(X) IMPLIES X |> mu(G)
fixed_point: LEMMA G(mu(G)) = mu(G)
weakest_fixed_point: LEMMA G(X) = X IMPLIES  X |> mu(G)
```

5.4. Machine-assisted Proof of Properties

So far, we have formalised our denotational semantic model and process semantics in PVS. In this section, we apply the PVS type checker to validating the consistency of the denotational semantics and the PVS prover to proving essential laws of our formalisations so as to check the correctness of our encoding.

The type checker in PVS analyses the theory for semantic consistency [OSRSC01]. It usually checks the semantic constraints, determines the types of expressions, and resolves names. After typechecking, proof obligations (TCCs) are generated which are mostly related to predicate subtypes and termination in the recursive definitions. In our work, we have discharged 53 TCCs for the process semantics theory. These TCCs are mainly from the subtypes requiring that every CSP# process satisfies the healthiness property and terminations of recursive definitions such as trace synchronisation function.

Based on our semantic formalisation in PVS, we can derive many important properties. We have machine-assisted proved a set of important laws that are essential in the verification of CSP# programs. Regarding the properties of healthiness conditions, we have proved that conditions `R1`, `R2`, `R3`, `CSP1`, and `CSP2` are idempotent and commutative. For example, the following PVS proof script is used to prove the commutativity property of `R1` and `CSP1`.

```
R1_CSP1_commutative :

  |-------
1   FORALL (P: Relation): CSP1(R1(P)) = R1(CSP1(P))

Rule? (skolem!)
Skolemizing,
this simplifies to:
R1_CSP1_commutative :

  |-------
1   CSP1(R1(P!1)) = R1(CSP1(P!1))

Rule? (expand* "CSP1" "R1")
Expanding the definition(s) of (CSP1 R1),
this simplifies to:
R1_CSP1_commutative :

  |-------
1   ({pre: AB |
          (NOT pre`ok AND pre`tr <= pre`tr1) OR P!1(pre) AND pre`tr <= pre`tr1})
      =
      ({pre_1: AB |
          ((NOT pre_1`ok AND pre_1`tr <= pre_1`tr1) OR P!1(pre_1)) AND pre_1`tr <= pre_1`tr1})

Rule? (apply-extensionality :hide? t)
```

```
Applying extensionality,
this simplifies to:
R1_CSP1_commutative :

  |-------
1   ((NOT x!1‘ok AND x!1‘tr <= x!1‘tr1) OR P!1(x!1) AND x!1‘tr <= x!1‘tr1)
       =
       (((NOT x!1‘ok AND x!1‘tr <= x!1‘tr1) OR P!1(x!1)) AND x!1‘tr <= x!1‘tr1)
Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

In the above proof script, command `skolem!` introduces Skolem constant for the universally quantified variable `P` in the lemma, command `expand*` expands the definitions of `CSP1` and `R1`, command `apply-extensionality :hide? t` uses extensionality to prove equality, and command `grind` installs rewrites and repeatedly applies simplification.

From these lemmas, we show that our formalisation of condition `H` is idempotent and every CSP# process is healthy. We have also proved that some important algebraic laws of CSP# processes: law (**guard-3**) is valid; internal choice is commutative and idempotent; etc.

6. Related Work

The denotational semantics of CSP has been defined using two approaches. On one hand, Roscoe [Ros97] and Hoare [Hoa85] provided a *trace* model, a *stable-failures* model and a *failures-divergences* model for CSP processes. In the trace model, every process is mapped to a set of traces which capture sequences of event occurrences during the process execution. In the stable-failures model, every process is mapped to a set of pairs, and each pair consists of a trace and a refusal. In the failures-divergences model, every process is mapped to a pair, where one component is a set of traces that can lead to divergent behaviours, and the other component contains all stable failures which are all pairs, and each pair is in the form of a trace and a refusal. On the other hand, Hoare and He [HH98] defined a denotational semantics for CSP processes using the UTP theory. Each process is formalised as a relation between an initial observation and a subsequent observation; such relations are represented as predicates over observational variables which record process stability, termination, traces and refusals *before* or *after* the observation. Cavalcanti and Woodcock [CW06] related the UTP theory of CSP to the *failures-divergences* model of CSP.

The aforementioned denotational semantics for CSP does not deal with data aspects. To solve this problem, several attempts have been made to provide the denotational semantics for languages which integrate CSP with state-based notations. For example, Oliveira *et al.* [OCW09] presented a denotational semantics for Circus based on a UTP theory. The proposed semantics includes two parts: one is for Circus actions, guarded commands, etc., and the other is for Circus processes which contain an encapsulated state, a main action, etc. However, this proposed semantics assumes that the sets of variables in processes shall be *disjoint* when those processes run in parallel or interleaving. Qin *et al.* [QDC03] formalised the denotational semantics of Timed Communicating Object Z (TCOZ) based on the UTP framework. Their unified semantic model can deal with channel-based and sensor/actuator-based communications [MD99], although shared variables in TCOZ are restricted to only sensors/actuators.

There exists some work on shared-variable concurrency. Brooks [Bro96] defined a denotational semantics for a shared-variable parallel language, where the semantic model considers state transitions only, and thus cannot be directly applied to communicating processes. Zhu *et al.* [ZBH01] derived a denotational semantics from the proposed operational semantics for the hardware description language Verilog. In addition, they [ZHB08] derived the denotational semantics from the algebraic semantics for Verilog to explore the equivalence of two semantic models. Recently, they [ZYH+12] proposed a probabilistic language *PTSC* which integrates probability, time and shared-variable concurrency. The operational semantics of PTSC is explored and a set of algebraic laws are presented via bisimulation. Furthermore, a denotational semantics using the UTP approach [ZQHB09] is derived from the algebraic laws based on the head normal form of PTSC constructs. These semantic models lack expressive power to capture more complicated system behaviours like channel-based communications.

Besides the above approaches on defining denotational semantics, considerable effort has been made around the encoding of various CSP models. Camilleri [Cam90] encoded the trace model of CSP and later a variation of the failures-divergences model [Cam91] into the HOL system [Gor88]. Dutertre and Schnei-

der [DS97] formalised the trace model of CSP in PVS, tailored to reason about security protocols; Wei and Heather [WH05] extended this formalisation to the stable-failures model in order to verify liveness properties. Tej and Wolff [TW97] encoded the failures-divergences model in Isabelle/HOL. Isobe and Roggenbach [IR05, IR08] improved this work with tool support from CSP-Prover which handles more CSP models including trace model, stable-failures model and stable-revivals model [SRI09]. However, all the above formalisation of various CSP models lacks the support of complex data.

There exists other research on encoding denotational semantics of integrated languages with CSP. For example, Oliveira *et al.* [OCW06] presented the encoding of the UTP semantics of Circus in ProofPower-Z [Pro]; the formalised semantics is defined using a set-based theory. Moreover, the machine-assisted proof of various refinement laws was reported in [OCW09]. Wei *et al.* [WWB09] encoded the UTP semantics of Timed Circus in PVS, where the formalisation of time operators Delay, Timeout and Deadline was presented. Our work follows in a similar way of their encoding, but covers the formalisation of sequential programs such as assignments, sequential composition on shared variables, while Timed Circus supports assignments on local variables only. Moreover, we formalise different event types covering both event synchronisation and pairwise handshake through synchronous channels.

This work is also related to research on formalisation of UTP theories. Feliachi *et al.* [FGW10] formalised a part of UTP theory in Isabelle/HOL including theories of alphabetised relations and designs. Recently Foster and Woodcock [FW13] improved the encoding of UTP theory, by defining a unified type for predicates and supporting more operators and meta-theoretic proofs; their Isabelle/UTP currently only supports theories of relations and designs, and it lacks the support of more UTP theories like theories of reactive processes (e.g., healthiness conditions **R1**, **R2** and **R3**). Nonetheless, Isabelle/UTP provides a platform for encoding the UTP semantics for specification languages like CSP.

7. Conclusion and Future Work

In this article, we proposed an observation-oriented semantics for the CSP# language based on the UTP framework. The formalised semantics covers different types of concurrency, i.e., communications and shared variable parallelism. In addition, a set of algebraic laws were proposed based on the denotational model for communicating processes involving shared variables. Furthermore, we encoded the proposed semantics into the PVS theorem prover. The consistency of encoded semantics was validated by proving the TCCs generated from typechecking. Based on the encoding, we also proved properties of healthiness conditions and algebraic laws related to process definitions in PVS.

There are several directions of the future work. One direction is to extend our proposed denotational semantic model to cover more system behaviours such as timed aspects and probability [SLS$^+$11, DSC$^+$15, SLH$^+$17]; for example, *Stateful Timed CSP* [SLD$^+$13] extends CSP# with time process constructs like *timeout* and *deadline*. Another direction is to enhance our verification framework by validating more complex theories (e.g., monotonicity property and algebraic laws in Section 4), and further to apply our framework to verifying real-world case studies, especially systems of infinite states, by leveraging the induction technique which complements model checking techniques.

Acknowledgements

The authors would like to thank Prof. Jim Woodcock for insightful comments on the denotational semantics of CSP#. This research is partially supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate, and National Research Foundation (No. NRF2015NCR-NCR003-003), Singapore. This work is partially supported by Science and Technology Commission of Shanghai Municipality Projects (No. 15511104700 and No. 16DZ1100600), Shanghai SHEITC Project (No. 160602), National Natural Science Foundation of China (NSFC 61402176 and 61602177).

References

[BBC+96] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, H. Herbelin, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.1*. INRIA-Rocquencourt-CNRS-ENS Lyon, 1996.

[Bro96] Stephen D. Brookes. Full Abstraction for a Shared-Variable Parallel Language. *Information and Computation*, 127(2):145–163, 1996.

[Cam90] Albert John Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transations on Software Engineering*, 16(9):993–1004, 1990.

[Cam91] Albert John Camilleri. A Higher Order Logic Mechanization of the CSP Failure-Divergence Semantics. In *IV Higher Order Workshop, Banff 1990*, pages 123–150. Springer London, 1991.

[CH09] Robert Colvin and Ian J. Hayes. CSP with Hierarchical State. In *7th International Conference on Integrated Formal Methods (IFM'09)*, volume 5423 of *Lecture Notes in Computer Science*, pages 118–135. Springer, 2009.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

[CW06] Ana Cavalcanti and Jim Woodcock. A Tutorial Introduction to CSP in *Unifying Theories of Programming*. In *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 220–268. Springer, 2006.

[Dek] Dekker's algorithm. `https://en.wikipedia.org/wiki/Dekker%27s_algorithm`.

[Dij68] Edsger W. Dijkstra. Cooperating sequential processes. In *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[DS97] Bruno Dutertre and Steve Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 1997.

[DSC+15] Jin Song Dong, Ling Shi, Le Vu Nguyen Chuong, Kan Jiang, and Jing Sun. Sports Strategy Analytics Using Probabilistic Reasoning. In *20th International Conference on Engineering of Complex Computer Systems, (ICECCS)*, pages 182–185, 2015.

[FGW10] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Unifying Theories in Isabelle/HOL. In *Third International Symposium on Unifying Theories of Programming (UTP'10)*, volume 6445 of *Lecture Notes in Computer Science*, pages 188–206. Springer, 2010.

[Fis97] Clemens Fischer. Combining Object-Z and CSP. In *FBT*, pages 119–128. GMD-Forschungszentrum Informationstechnik GmbH, 1997.

[FW13] Simon Foster and Jim Woodcock. Unifying Theories of Programming in Isabelle. In *ICTAC Training School on Software Engineering*, volume 8050 of *Lecture Notes in Computer Science*, pages 109–155. Springer, 2013.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Gor88] Michael J. C. Gordon. HOL: a proof generating system for Higher Order Logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Springer US, 1988.

[GS97] Andy Galloway and Bill Stoddart. An Operational Semantics for ZCCS. In *1st International Conference on Formal Engineering Methods (ICFEM'97)*, pages 272–282. IEEE Computer Society, 1997.

[HH98] C.A.R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[HHH+87] C. A. R. Hoare, Ian J. Hayes, Jifeng He, Carroll Morgan, A. W. Roscoe, Jeff W. Sanders, Ib Holm Sørensen, J. Michael Spivey, and Bernard Sufrin. Laws of Programming. *Communications of the ACM*, 30(8):672–686, 1987.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[IR05] Yoshinao Isobe and Markus Roggenbach. A Generic Theorem Prover of CSP Refinement. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2005.

[IR08] Yoshinao Isobe and Markus Roggenbach. Proof Principles of CSP CSP-Prover in Practice. In *First International Conference on Dynamics in Logistics (LDIC'07)*, pages 425–442. Springer Berlin Heidelberg, 2008.

[MD99] Brendan P. Mahony and Jin Song Dong. Sensors and Actuators in TCOZ. In *World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, pages 1166–1185. Springer Berlin Heidelberg, 1999.

[MD00] Brendan P. Mahony and Jin Song Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.

[MD02] Brendan P. Mahony and Jin Song Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspect of Computing*, 13(2):142–160, 2002.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[OCW06] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying Theories in Proofpower-Z. In *First International Symposium on Unifying Theories of Programming (UTP'06)*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2006.

[OCW09] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3–32, 2009.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer-Verlag, 1992.

[OSRSC01] Sam Owre, Natarajan Shankar, John Rushby, and David WJ Stringer-Calvert. *PVS System Guide*. SRI International, November 2001.

[Pau94] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[PDvHR96] H. Pfeifer, A. Dold, F. W. von Henke, and H. Rueß. Mechanized Semantics of Simple Imperative Programming Constructs. Technical Report 96-11, Universität Ulm, Fakultät für Informatik, 1996.

[Pro] ProofPower. `http://www.lemma-one.com/ProofPower/index/index.html`.

[QDC03] Shengchao Qin, Jin Song Dong, and Wei-Ngan Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2003.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[SLD+13] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Transactions on Software Engineering and Methodology*, 22(1):3:1–3:29, 2013.

[SLDC09] Jun Sun, Yang Liu, Jin Song Dong, and Chunqing Chen. Integrating Specification and Programs for System Modeling and Verification. In *The 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09)*, pages 127–135. IEEE Computer Society, 2009.

[SLDP09] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards Flexible Verification under Fairness. In *21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[SLH+17] Ling Shi, Shuang Liu, Jianye Hao, Jun Yang Koh, Zhe Hou, and Jin Song Dong. Towards Solving Decision Making Problems Using Probabilistic Model Checking. In *22nd International Conference on Engineering of Complex Computer Systems, (ICECCS)*, pages 150–153, 2017.

[SLS+11] Jun Sun, Yang Liu, Songzheng Song, Jin Song Dong, and Xiaohong Li. PRTS: An Approach for Model Checking Probabilistic Real-Time Hierarchical Systems. In *13th International Conference on Formal Engineering Methods (ICFEM'11)*, volume 6991 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2011.

[SLS+12] Ling Shi, Yang Liu, Jun Sun, Jin Song Dong, and Gustavo Carvalho. An Analytical and Experimental Comparison of CSP Extensions and Tools. In *14th International Conference on Formal Engineering Methods (ICFEM'12)*, volume 7635 of *Lecture Notes in Computer Science*, pages 381–397. Springer, 2012.

[Smi97] Graeme Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer, 1997.

[SORSC01] Natarajan Shankar, Sam Owre, John Rushby, and David WJ Stringer-Calvert. *PVS Prover Guide*. SRI International, November 2001.

[SRI09] D. Gift Samuel, Markus Roggenbach, and Yoshinao Isobe. The stable revivals model in CSP-Prover. *Electronic Notes in Theoretical Computer Science*, 250(2):119–134, 2009.

[ST05] Steve Schneider and Helen Treharne. CSP Theorems for Communicating B Machines. *Formal Aspects of Computing*, 17(4):390–422, 2005.

[SZL+13] Ling Shi, Yongxin Zhao, Yang Liu, Jun Sun, Jin Song Dong, and Shengchao Qin. A UTP Semantics for Communicating Processes with Shared Variables. In *15th International Conference on Formal Engineering Methods (ICFEM'13)*, volume 8144 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2013.

[TA97] Kenji Taguchi and Keijiro Araki. The State-Based CCS Semantics for Concurrent Z Specification. In *1st International Conference on Formal Engineering Methods (ICFEM'97)*, pages 283–292. IEEE Computer Society, 1997.

[TW97] Haykal Tej and Burkhart Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337. Springer, 1997.

[WC02] Jim Woodcock and Ana Cavalcanti. The Semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.

[WH05] Kun Wei and James Heather. Embedding the stable failures model of CSP in PVS. In *5th International Conference on Integrated Formal Methods (IFM'05)*, volume 3771 of *Lecture Notes in Computer Science*, pages 246–265. Springer, 2005.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):19:1–19:36, 2009.

[WWB09] Kun Wei, Jim Woodcock, and Alan Burns. Embedding the Timed Circus in PVS. Technical report, University of York, 2009.

[ZBH01] Huibiao Zhu, Jonathan P. Bowen, and Jifeng He. From Operational Semantics to Denotational Semantics for Verilog. In *Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Lecture Notes in Computer Science*, pages 449–466. Springer, 2001.

[ZHB08] Huibiao Zhu, Jifeng He, and Jonathan P. Bowen. From algebraic semantics to denotational semantics for Verilog. *ISSE*, 4(4):341–360, 2008.

[ZQHB09] Huibiao Zhu, Shengchao Qin, Jifeng He, and Jonathan P. Bowen. PTSC: probability, time and shared-variable concurrency. *ISSE*, 5(4):271–284, 2009.

[ZYH+12] Huibiao Zhu, Fan Yang, Jifeng He, Jonathan P. Bowen, Jeff W. Sanders, and Shengchao Qin. Linking Operational Semantics and Algebraic Semantics for a Probabilistic Timed Shared-Variable Language. *The Journal of Logic and Algebraic Programming*, 81(1):2–25, 2012.