View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Institutional Knowledge at Singapore Management University

Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems

School of Information Systems

11-2013

# A UTP semantics for communicating processes with shared variables

Ling SHI

Yongxin ZHAO

Yang LIU

Jun SUN
*Singapore Management University*, junsun@smu.edu.sg

Jin Song DONG

*See next page for additional authors*

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the Software Engineering Commons

## Citation

SHI, Ling; ZHAO, Yongxin; LIU, Yang; SUN, Jun; DONG, Jin Song; and QIN, Shengchao. A UTP semantics for communicating processes with shared variables. (2013). *Proceedings of the 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 - November 1*. 215-230. Research Collection School Of Information Systems.
**Available at:** https://ink.library.smu.edu.sg/sis_research/4999

This Conference Proceeding Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

## Author

Ling SHI, Yongxin ZHAO, Yang LIU, Jun SUN, Jin Song DONG, and Shengchao QIN

This conference proceeding article is available at Institutional Knowledge at Singapore Management University: https://ink.library.smu.edu.sg/sis_research/4999

# A UTP Semantics for Communicating Processes with Shared Variables*

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

[1] National University of Singapore
[2] Nanyang Technological University, Singapore
[3] Singapore University of Technology and Design
[4] Teesside University, 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 paper, 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 *hybrid* traces which consist of event-state pairs for recording process behaviours. We also define refinement to check process equivalence and present a set of algebraic laws which are established based on our denotational semantics. Our approach thus provides a rigorous means for reasoning about the correctness of CSP# process behaviours. We further derive a *closed* semantics by focusing on special types of hybrid traces; this closed semantics can be linked with existing CSP# operational semantics.

## 1 Introduction

Communicating Sequential Processes (CSP) [6], 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 [18]. 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 functional aspects. To solve this problem, considerable efforts on enhancing CSP with data aspects have been made. One of the approaches is to integrate CSP (CCS) with state-based specification languages, such as Circus [8], CSP-OZ [4,13], TCOZ [9], CSP$_\sigma$ [3], CSP||B [11], and CCS+Z [5,16].

Inspired by the related works, CSP# [14] has been proposed to specify concurrent systems which involve *shared variables*. It combines the state-based program with the event-based specification by introducing non-communicating events to associate state transitions. CSP# integrates CSP-like compositional operators with sequential program

* This work is partially supported by project "ZJURP1100105" from Singapore University of Technology and Design.

L. Groves and J. Sun (Eds.): ICFEM 2013, LNCS 8144, pp. 215–230, 2013.
© Springer-Verlag Berlin Heidelberg 2013

constructs such as assignments and while loops, for the purpose of expressive modelling and efficient system verification[1]. Besides, CSP# is supported by the PAT model checker [15] and has been applied to a number of systems available at the PAT website (`www.patroot.com`).

Sun *et al.* presented an operational semantics of CSP# [14], which interprets the behaviour of CSP# models using labelled transition systems (LTS). Based on this semantics, model checking CSP# models becomes possible. Nevertheless, the suggested operational semantics is not fully abstract; two behaviourally equivalent processes with respect to the operational semantics may behave differently under some process context which involves shared variables, for instance. In other words, the operational semantics of CSP# is not compositional and lacks the support of compositional verification of process behaviours. Thus there is a need for a compositional semantics to explain the notations of the CSP# language.

*Related Work.* The denotational semantics of CSP has been defined using two approaches. On one hand, Roscoe [10] and Hoare [6] 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 the (extension-closed) 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 defined a denotational semantics for CSP processes using the UTP theory [7]. 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 [2] presented an approach to relate the UTP theory of CSP to the *failures-divergences* model of CSP.

The original denotational semantics for CSP does not deal with complex data aspects. To solve this problem, much work has been done to provide the denotational semantics for languages which integrate CSP with state-based notations. For example, Oliveira *et al*. presented a denotational semantics for Circus based on a UTP theory [8]. 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 running in parallel or interleaving. Qin *et al*. formalised the denotational semantics of Timed Communicating Object Z (TCOZ) [9] based on the UTP framework. Their unified semantic model can deal with channel-based and sensor/actuator-based communications. However, shared variables in TCOZ are restricted to only sensors/actuators.

There exists some work on shared-variable concurrency. Brooks defined a denotational semantics for a shared-variable parallel language [1]. The semantic model only

[1] Most integrated formalisms are too expressive to have an automated supporting tool. CSP# combines CSP with C#-like program instead of Z.

considers state transitions, and it cannot be directly applied to the semantics of communicating processes. Zhu *et al*. derived an denotational semantics from the proposed operational semantics for the hardware description language Verilog [19]. In addition, they derived the denotational semantics from the algebraic semantics for Verilog to explore the equivalence of two semantic models [20]. Recently, they proposed a probabilistic language *PTSC* which integrates probability, time and shared-variable concurrency [22]. 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 is derived from the algebraic laws based on the head normal form of PTSC constructs [21]. These semantic models lack expressive power to capture more complicated system behaviours like channel-based communications.

The above existing work cannot be applied to define the denotational semantics of CSP# which involves global shared variables. In this paper, we present an observation-oriented denotational semantics for the CSP# language based on the UTP 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 deduces algebraic laws describing the properties of CSP# processes. To deal with shared variables, we lift traditional event-based traces into *hybrid* traces (consisting of event-state pairs) for recording process behaviours. To handle different types of synchronisation in CSP# (i.e., event-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 concurrency behaviours between event/channel-based communications and global shared variables.

*Contribution.* We highlight our contributions by the three points below.

- The proposed semantic model deals with not only communicating processes, but also shared variables. It can model both event-based synchronisation and synchronised handshake over channels. Moreover, our model can be adapted/enhanced to define the denotational semantics for other languages which possess similar concurrency mechanisms.
- The semantics of processes can serve as a theoretical foundation to develop mechanical verification for CSP# specifications, for example, to check process equivalence based on our definition of process refinement, using conventional generic theorem provers like PVS. In addition, the proposed algebraic laws can act as auxiliary reasoning rules to improve verification automation.
- A *closed* semantics can be derived from our open denotational semantics by focusing on special types of hybrid traces. The closed semantics can be linked with the CSP# operational semantics in [14].

The remainder of the paper is organised as follows. Section 2 introduces the syntax of the CSP# language with informal descriptions. 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 presents a closed semantics derived from the open semantics. Section 6 concludes the paper with future work.

## 2 The CSP# Language

***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* $x = 2$. A channel is declared using keyword *channel* with a name, e.g., *channel ch*. Notice that we use T to denote the types of variables and channel messages and T will be used in Section 3.1.1. A process is specified in the form of $Proc(i_1, i_2, \ldots, i_n) = ProcExp$, where *Proc* is the process name, $(i_1, i_2, \ldots, i_n)$ is an optional list of process parameters and *ProcExp* is a process expression. The BNF description of *ProcExp* is shown below with short descriptions.

| | |
|---|---|
| $P ::= Stop \mid Skip$ | – primitives |
| $\mid\ a \rightarrow P$ | – event prefixing |
| $\mid\ ch!exp \rightarrow P \mid ch?m \rightarrow P(m)$ | – channel output/input |
| $\mid\ e\{prog\} \rightarrow P$ | – data operation prefixing |
| $\mid\ [b]P$ | – state guard |
| $\mid\ P \,\square\, Q \mid P \sqcap Q$ | – external/internal choices |
| $\mid\ P;\ Q$ | – sequential composition |
| $\mid\ P \setminus X$ | – hiding |
| $\mid\ P \parallel Q \mid P \,\|\|\|\, Q$ | – parallel/interleaving |
| $\mid\ p \mid \mu p \bullet P(p)$ | – recursion |

where $P$ and $Q$ are processes, $a$ is an action, $e$ is a non-communicating event, $ch$ is a channel, $exp$ is an arithmetic expression, $m$ is a bounded variable, $prog$ is a sequential program updating global shared variables, $b$ is a Boolean expression, and $X$ is a set of actions. In addition, the syntax of $prog$ is illustrated as follows.

| | |
|---|---|
| $prog ::= x = exp$ | – assignment |
| $\mid$ $prog_1;\ prog_2$ | – composition |
| $\mid$ if $b$ then $prog_1$ else $prog_2$ | – conditional |
| $\mid$ while $b$ do $prog$ | – iteration |

In CSP#, channels are *synchronous* and their communications are achieved by a handshaking mechanism. Specifically, a process $ch!exp \rightarrow P$ which is ready to perform an output through $ch$ will be enabled if another process $ch?m \rightarrow P(m)$ is ready to perform an input through the same channel $ch$ simultaneously, and *vice versa*. In process $e\{prog\} \rightarrow P$, $prog$ is executed *atomically* with the occurrence of $e$. Process $[b]P$ waits until condition $b$ becomes *true* and then behaves as $P$. There are two types of choices in CSP#: external choice $P \,\square\, Q$ is resolved only by the occurrence of a *visible* event, and internal choice $P \sqcap Q$ is resolved non-deterministically. In process $P \parallel Q$, $P$ and $Q$ run in parallel, and they synchronise on common communication events. In contrast, in process $P \,|||\, Q$, $P$ and $Q$ run independently (except for communications through synchronous channels). Detailed descriptions of the CSP# syntax can be found in [14].

***Concurrency.*** As mentioned earlier, concurrent processes in CSP# can communicate through shared variables or event/channel-based communications.

Shared variables in CSP# are globally accessible, namely, variables can be read and written by different (parallel) processes. They can be used in guard conditions, sequential programs associated with non-communicating events, 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 more than one 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 is different from $\text{CSP}_M$ where multi-way synchronisation between many sender and receiver processes is allowed [10].

## 3 The Observation-Oriented Semantics for CSP#

### 3.1 UTP Semantic Model for CSP#

UTP [7] uses relations as a unifying basis to define denotational semantics for programs across different programming paradigms. Theories of programming paradigms are differentiated by their *alphabet*, *signature* and a selection of laws called *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 identify valid predicates that characterise a theory.

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; 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.

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. To address this challenge, we blend communication events with states containing shared variables. Namely, we introduce *hybrid* 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 behaviour. Some of them (i.e., $ok$, $ok'$, $wait$, $wait'$, $ref$, and $ref'$) are similar to those in the UTP theory for CSP [7]. The key difference is that the event-based traces in CSP are changed to hybrid traces consisting of event-state 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*′: 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 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*.
- *tr*, *tr*′: $\mathrm{seq}((\mathsf{S} \times \mathsf{S}^{\perp}) \cup (\mathsf{S} \times \mathsf{E}))$ record a sequence of observations (state pairs or communication events) on the interaction of processes with the global environment.
  - $\mathsf{S}$ is the set of all possible mappings (states), and a state $s : \mathsf{VAR} \to \mathsf{T}$ is a function which maps global shared variables $\mathsf{VAR}$ into values of $\mathsf{T}$.
  - $\mathsf{E}$ is the set of all possible events, including actions, channel inputs/outputs and synchronous channel communications. Note that non-communicating events are excluded from the set.
  - $\mathsf{S} \times \mathsf{S}^{\perp}$ 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. Remark that the state pair is used to record the observation for the sequential program.
  - $\mathsf{S} \times \mathsf{E}$ 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 $\phi$ 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 function **R1**. We use predicate *P* to represent the semantics of the CSP# process below.

**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 an waiting state from the final state. A process cannot start if its previous process has not finished, or otherwise, the values of all observational variables are unchanged, characterised by function **R3**.

**R3**: $\mathbf{R3}(P) \;=\; I\!I \lhd wait \rhd P$

where $P \lhd b \rhd Q \mathrel{\widehat{=}} b \wedge P \vee \neg b \wedge Q$ and $I\!I \;\mathrel{\widehat{=}}\; (\neg ok \wedge tr \leq tr') \vee (ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref)$. Here $I\!I$ 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 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**: $\mathbf{CSP2}(P) \;=\; P;\ (ok \Rightarrow ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref)$

The behaviour of a process does not depend on the initial value of its refusal, captured by function **CSP3**.

**CSP3**: $\mathbf{CSP3}(P) \;=\; Skip;\ P$

Similarly, when a process terminates or diverges, the value of its final refusal is irrelevant, characterised by function **CSP4**.

**CSP4**: $\mathbf{CSP4}(P) \;=\; P;\ Skip$

If a deadlocked process refuses some set of events offered by its environment, then it would still be deadlocked in an environment that offers even fewer events, captured by function **CSP5**

**CSP5**: $\mathbf{CSP5}(P) \;=\; P \;|||\; Skip$

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

$$\mathbf{H} = \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3} \circ \mathbf{CSP1} \circ \mathbf{CSP2} \circ \mathbf{CSP3} \circ \mathbf{CSP4} \circ \mathbf{CSP5}$$

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 [2,7].

### 3.2 Process Semantics

In this section, we construct an observation-oriented semantics for all CSP# process operators based on our proposed UTP semantic model for CSP#. The semantics is defined in an open environment; namely, a process may be interfered with by the environment. In Section 3.1.1, we have defined a hybrid trace to record the potential events and state transitions in which a process $P$ may engage; for example, the trace $tr' = \langle (s_1, s_1') \rangle \frown \langle (s_2, a_2) \rangle$ 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 first illustrate our semantic definitions of three important process operators: synchronous channel output/input, data operation prefixing, and parallel composition. These three process operators are non-trivial and frequently used in complex concurrent systems with the involvement of channel-based communications and shared variables. We further present the semantics of other process operators and refinement at the end; detailed semantic definitions of the complete CSP# language are available in our technical report [12].

### 3.2.1 Synchronous Channel Output/Input

In CSP#, messages can be sent/received synchronously through channels. The synchronisation is pair-wise, 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.

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

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. Since the environment may interfere with the process behaviour and make a transition on the shared variable states, we use state *s* to denote the initial state before the observation (also named pre-state). 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 available in [12]. After the output occurs, the process behaves as *P*. Note that the semantics of sequential composition "; " is defined in Section 3.2.4.

$$ch?m \rightarrow P(m) \mathrel{\widehat{=}} \exists\, v \in \mathsf{T} \bullet \left(\mathbf{H}\left(ok' \wedge \begin{pmatrix} ch! \notin ref' \wedge tr' = tr \\ \lhd wait' \rhd \\ \exists\, s \in \mathsf{S} \bullet tr' = tr \frown \langle (s, ch?v) \rangle \end{pmatrix}\right) ;\ P(v)\right)$$

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 receives 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 *P*; namely, it becomes constant-like and its value is replaced by value *v*.

### 3.2.2 Data Operation Prefixing

In CSP#, sequential programs are executed atomically together with the occurrence of an event, called data operation. The updates on shared variables are observed after the execution of all programs as illustrated below.

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

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. In our definition,

the non-communicating event is not recorded in the trace since such an event would not synchronise with other events; instead, its effect can be described by the updates on variable states. Thus the non-communicating event is used as a label to indicate the updates on shared variables. 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 programs by structured induction [17] as follows.

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

where, $F(X) \mathrel{\widehat{=}} \mathsf{if}\ b\ \mathsf{then}\ prog;\ X\ \mathsf{else}\ skip$, $\mu X \bullet F(X) \mathrel{\widehat{=}} \bigcap_n F^n(true)$, $\mathcal{C}[\![skip]\!] = \{(s, s) \mid s \in \mathcal{S}\}$, and $\mathcal{C}[\![true]\!] = \{(s, s') \mid s \in \mathcal{S}, s' \in \mathcal{S}^{\bot}\}$.

The data operation prefixing process $e\{prog\} \rightarrow P$ is thus defined as sequential composition of data operation and $P$.

$$e\{prog\} \rightarrow P \mathrel{\widehat{=}} (e\{prog\} \rightarrow Skip);\ P$$

### 3.2.3 Parallel Composition

The parallel composition $P \parallel Q$ 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 $\|_M$ in UTP [7], 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 $M(X)$ to model the merge operation. The set $X$ contains common actions of both processes (denoted by set $X_1$) and all synchronous channel inputs and outputs (denoted by set $X_2$). Namely,

$$P \parallel Q \mathrel{\widehat{=}} \begin{pmatrix} P[0.ok, 0.wait, 0.ref, 0.tr/ok', wait', ref', tr'] \wedge \\ Q[1.ok, 1.wait, 1.ref, 1.tr/ok', wait', ref', tr'] \end{pmatrix};\ M(X)$$

where

$$M(X) \mathrel{\widehat{=}} \begin{pmatrix} (ok' = 0.ok \wedge 1.ok) \wedge \\ (wait' = 0.wait \vee 1.wait) \wedge \\ (ref' = (0.ref \cap 1.ref \cap X_2) \cup ((0.ref \cup 1.ref) \cap X_1) \\ \qquad \cup ((0.ref \cap 1.ref) - X_1 - X_2)) \\ (tr' - tr \in (0.tr - tr \parallel_X 1.tr - tr)) \end{pmatrix};\ Skip$$

Our defined predicate $M(X)$ 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 ouputs/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 actions that are not in the set $X_1$ 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 $\|_X$ as elaborated below.

Function $\|_X$ models how to merge two individual traces into a set of all possible traces; there are 9 cases from 6 groups. In the following definitions, $s$, $s'$, $s_1$, $s_1'$, $s_2$, $s_2'$ are representative elements of variable states, $a$, $a_1$, $a_2$ are representative elements of actions, $ch$ is a representative element of channel names, and $v$ is a value with type $\mathsf{T}$.

- When one of the input traces is empty, (1) if both input traces are empty, the result is a set of an empty sequence (denoted by **case-1**); (2) if only one input trace is empty, the result is determined based on the first observation of that non-empty trace: (i) if that observation is an action in the set $X$ 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-2**); if the first observation is (ii) a channel input/output/communication (**case-3**) or (iii) a state pair (**case-4**), then the observation is recorded in the merged trace.

  **case-1** $\langle\rangle \|_X \langle\rangle = \{\langle\rangle\}$

  **case-2** $\langle(s,a)\rangle \frown t \|_X \langle\rangle = \begin{cases} \{\langle\rangle\} & \text{if } a \in X \\ \{\langle(s,a)\rangle \frown l \mid l \in t \|_X \langle\rangle\} & \text{otherwise} \end{cases}$

  **case-3** $\langle(s,h)\rangle \frown t \|_X \langle\rangle = \{\langle(s,h)\rangle \frown l \mid l \in t \|_X \langle\rangle\}$, where $h \in \{ch?v, ch!v, ch.v\}$

  **case-4** $\langle(s,s')\rangle \frown t \|_X \langle\rangle = \{\langle(s,s')\rangle \frown l \mid l \in t \|_X \langle\rangle\}$
- When a communication is over a synchronous channel, (1) if the first observations of two input traces match (see Definition 1 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 $\mathcal{G}_2$), or 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 1 (Match).** *Given two pairs $p_1 = (s_1, h_1)$ and $p_2 = (s_2, h_2)$, we say that they are matched if both $s_1 = s_2$ and $\{h_1, h_2\} = \{ch?v, ch!v\}$ are satisfied, denoted as* $\mathsf{match}(p_1, p_2)$.

  **case-5** $\langle(s_1,h_1)\rangle \frown t_1 \|_X \langle(s_2,h_2)\rangle \frown t_2 = \begin{cases} \mathcal{G}_1 \cup \mathcal{G}_2 & \mathsf{match}((s_1,h_1),(s_2,h_2)) \\ \mathcal{G}_2 & \textit{otherwise} \end{cases}$

  where $h_1, h_2 \in \{ch?v, ch!v, ch.v\}$, $\mathcal{G}_1 \mathrel{\widehat{=}} \{\langle(s_1,ch.v)\rangle \frown l \mid l \in t_1 \|_X t_2\}$, and $\mathcal{G}_2 \mathrel{\widehat{=}} \{\langle(s_1,h_1)\rangle \frown l \mid l \in t_1 \|_X \langle(s_2,h_2)\rangle \frown t_2\} \cup \{\langle(s_2,h_2)\rangle \frown l \mid l \in \langle(s_1,h_1)\rangle \frown t_1 \|_X t_2\}$.
- When two actions ($a_1$ and $a_2$) are synchronised, there are five cases with respect to the initial states ($s_1$ and $s_2$) and actions from the first observation of two input traces: (1) both actions are in the set $X$ but different, (2) actions from $X$ are the same but under different pre-states, (3) actions from $X$ are the same and under the same pre-state, (4) one of the actions is not in $X$, and (5) both actions are not in $X$. As

shown in **case-6** below, the result is a set containing only an empty sequence for cases (1) and (2). A synchronisation occurs under case (3), although it is postponed to occur under case (4). Either action can occur for case (5).

**case-6** $\langle (s_1, a_1) \rangle \frown t_1 \parallel_X \langle (s_2, a_2) \rangle \frown t_2 =$

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

– When the merge operation is on an action $a$ and channel input $ch?v$, output $ch!v$, communication $ch.v$, or a post-state $s_2'$, (1) if $a$ is from the set $X$, 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-7** $\langle (s_1, a) \rangle \frown t_1 \parallel_X \langle (s_2, h) \rangle \frown t_2 = \begin{cases} \mathcal{G}_3 & \textit{if } a \in X \\ \mathcal{G}_3 \cup \mathcal{G}_4 & \textit{otherwise} \end{cases}$

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

– When the merge operation is over two state pairs or the operation is on a state pair and a channel input/output/communication, either observation from two processes can occur as only one process can update shared variable(s) at a time when processes run in parallel.

**case-8** $\langle (s_1, s_1') \rangle \frown t_1 \parallel_X \langle (s_2, h) \rangle \frown t_2 = \{\langle (s_1, s_1') \rangle \frown l \mid l \in t_1 \parallel_X \langle (s_2, h) \rangle \frown t_2\} \cup \{\langle (s_2, h) \rangle \frown l \mid l \in \langle (s_1, s_1') \rangle \frown t_1 \parallel_X t_2\}$ where $h \in \{s_2', ch?v, ch!v, ch.v\}$

– Finally , function $\parallel_X$ is symmetric.

**case-9** $t_1 \parallel_X t_2 = t_2 \parallel_X t_1$

### 3.2.4 Other Processes and Refinement

The semantics of other processes is the same as those counterparts in the CSP model [7] except state guard ($[b]P$) and interleaving ($P \mid\mid\mid Q$) due to the involvement of shared variables as we illustrate below.

For process $[b]P$, shared variables can be read in the Boolean expression $b$, and $b$ is evaluated simultaneously with the occurrence of the first event of process $P$. That is to say, the evaluation of $b$ is under the pre-state of the first observation of process $P$ ($\mathcal{B}[\![b]\!](\pi_1(head(tr' - tr)))$), and if the evaluation returns true, then the process behaves as $P$, or otherwise, the process behaves as process *Stop*. Here, function $\mathcal{B}$ defines the semantics of Boolean expressions, function $\pi_1$ selects the first element of a tuple, and function *head* returns the first element of a sequence.

Process $P \mid\mid\mid Q$ runs independently except for the communications through synchronous channels. Thus, we define the semantics of the interleaving operator in a similar way of handling parallel operator (in Section 3.2.3) except that the set $X$ contains only synchronous channel inputs/outputs.

$$P;\ Q \mathrel{\widehat{=}} \exists\, obs_0 \bullet (P[obs_0/obs'] \wedge Q[obs_0/obs])^2$$
$$[b]P \mathrel{\widehat{=}} P \triangleleft (tr < tr' \wedge \mathcal{B}(b)(\pi_1(head(tr' - tr))) = true) \triangleright Stop$$
$$P \mid\mid\mid Q \mathrel{\widehat{=}} P \parallel_{M(X)} Q$$

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 $\forall a, a', \cdots \bullet P \Rightarrow Q$, where $a, a', \cdots$ are all the variables of the alphabet, which must be the same for the specification and implementation. The universal quantification implication is usually denoted as $[P \Rightarrow Q]$. The definition of refinement in CSP# is given as below.

**Definition 2 (Refinement).** *Let $P$ and $Q$ be predicates for 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 $P = [x = 2]b \rightarrow Skip \ \Box\ [x \neq 2]c \rightarrow Skip$, and a process $Q = [x = 2]b \rightarrow Skip \ \Box\ [x \neq 2]d \rightarrow Skip$, 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 $P = e\{x = x + 1\} \rightarrow e\{x = x + 1\} \rightarrow Skip$, and a process $Q = e\{x = x + 2\} \rightarrow Skip$, the refinement $P \sqsupseteq Q$ does not hold.

**Definition 3 (Equivalence).** *For any two CSP# processes $P$ and $Q$, $P$ is equivalent to $Q$ if and only if $P \sqsupseteq Q \wedge Q \sqsupseteq P$.*

**Lemma 1.** *All process combinators defined in the CSP# language are monotonic.*

**Theorem 1.** *The open semantics of CSP# is compositional.*

The proofs of Lemma 1 and Theorem 1 are available in appendix.

## 4 Algebraic Laws

In this section, we present a set of algebraic laws concerning the distinct features of CSP#. All algebraic laws can be established based on our denotational model. That is to say, if the equality of two differently written 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 provide an easier

[2] The term *obs* represents the set of observational variables *ok*, *wait*, *tr*, and *ref*, as is the case of $obs_0$ and $obs'$.

way to prove process equivalence during the theorem proving procedures. Due to the space limitations, proofs that the algebraic laws are sound with respect to the denotational semantics are available in [12].

**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** $[\mathit{false}]P = \mathit{Stop}$
**guard - 4** $[\mathit{true}]P = P$
**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 $[\mathit{false}]P$ behaves like *Stop* because its guard can never be fired. **guard - 4** shows that process $[\mathit{true}]P$ always activates the process $P$.

**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** $P = \mathit{Skip};\ P$
**seq - 5** $P = P;\ \mathit{Skip}$
**seq - 1** shows that sequential composition is associative. **seq - 2, 3** show the distribution of sequential composition through external choice. **seq - 4, 5** show that process *Skip* is the left and right unit of sequential composition, respectively.

**Parallel Composition**
**par - 3** $\mathit{Skip} \parallel P = P = P \parallel \mathit{Skip}$
**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.

## 5 The Closed Semantics

So far, we have constructed an open semantics for CSP#. Namely, the denotational semantics is defined in an open environment. The interference by the environment is implicitly captured in the hybrid trace which collects the potential events or state transitions in which a process may engage. For example, given a trace $\langle(s_1, s_1')\rangle \frown \langle(s_2, e)\rangle$, the transition from state $s_1'$ to $s_2$ is implicit, and it is performed by the environment. In addition, the environment can change the states, so it is not necessary to ensure that state $s_1'$ is the same as $s_2$. Thus the system and environment alternate in making transitions. From Theorem 1, the open semantics maintains the compositionality of the processes. Therefore, it supports compositional verification of process behaviours.

However, if we look at it in another light, there is no need to retain all possible transitions from the environment if we have already built the model of the whole system or the behaviour of the environment has been modelled as a process. In this situation, we attempt to consider a closed semantics for the CSP# language. Fortunately, the closed

semantics does not need to be defined from the scratch; it can be generated from the open semantics. Thus, we first introduce the definition of closed traces to judge which trace exactly describes the process behaviour in a closed environment.

**Definition 4 (Closed Trace).** *A hybrid trace tr is closed, represented as* $\mathsf{cl}(tr)$*, if it satisfies the following two conditions.*

*(1) For any state pair which is not the last element in the trace, the post-state is passed as the pre-state of its immediate subsequent element, i.e.,* $\forall 0 \leq i < \#tr - 1, \exists s, s' \in \mathsf{S} \bullet (tr_i = (s, s') \Rightarrow s' = \pi_1(tr_{(i+1)}))$[3].

*(2) For any event which is not the last element in the trace, it should share the same pre-state with its immediate subsequent element, i.e.,* $\forall 0 \leq i < \#tr - 1, \exists s \in \mathsf{S}, a \in \mathsf{E} \bullet (tr_i = (s, a) \Rightarrow s = \pi_1(tr_{(i+1)}))$.

Informally speaking, a closed trace has this property: two adjacent elements in the trace are associated by a common state; the post-state of the former equals to the pre-state of the latter if the former is a state transition; the pre-state is shared if the former is an event. Note that every element in a hybrid trace has a pre-state but only the state transition possesses a post-state because the pre-state is not changed when an event occurs. Since the environment cannot update the shared state, a closed trace is identified as the behaviour of the process in the closed environment. For convenience, given a set of hybrid traces, denoted as the set $HT$, we define $\mathsf{CL}(HT)$ to represent the set of all closed traces in $HT$. Obviously, we have $\mathsf{CL}(HT) \subseteq HT$.

Now, we can generate the closed semantics (denoted by $[\![P]\!]_{closed}$) from the open semantics ($[\![P]\!]_{open}$) for any communicating process $P$. The relation between them is revealed by Definition 5.

**Definition 5 (Closed Semantics).** $[\![P]\!]_{closed} \mathrel{\widehat{=}} [\![P]\!]_{open} \wedge \mathsf{cl}(tr) \wedge \mathsf{cl}(tr')$

According to the open semantics, two processes that are semantically equivalent can generate the same traces $tr, tr'$. Further, any two closed traces generated from their open traces are the same. Thus the equality with respect to the open semantics is preserved by the closed semantics, which is shown in Theorem 2.

**Theorem 2.** $[\![P]\!]_{open} = [\![Q]\!]_{open} \Rightarrow [\![P]\!]_{closed} = [\![Q]\!]_{closed}$

However, we cannot imply that $[\![P]\!]_{open} = [\![Q]\!]_{open}$ is true when $[\![P]\!]_{closed} = [\![Q]\!]_{closed}$ holds. Furthermore, given that $[\![P]\!]_{closed} = [\![Q]\!]_{closed}$, the law $P \parallel R = Q \parallel R$ may be invalid; the compositionality fails in the closed semantics as shown by Example 1.

*Example 1.* Given a process $P = a\{x = 2\} \rightarrow ([x = 2]b \rightarrow Skip \,\Box\, [x \neq 2]c \rightarrow Skip)$, and a process $Q = a\{x = 2\} \rightarrow ([x = 2]b \rightarrow Skip \,\Box\, [x \neq 2]d \rightarrow Skip)$, the closed semantics of processes $P$ and $Q$ is the same, while their open semantics is not the same because after executing the event $a$, process $P$ may execute event $c$, and process $Q$ may execute event $d$ when the value of variable $x$ is not equal to 2 in their pre-states. Therefore, given a process $R = e\{x = 3\} \rightarrow Skip$, there is a case that after executing the events $a$ and $e$ sequentially, process $P \parallel R$ will execute event $c$ while process $Q \parallel R$ will execute event $d$, and thus the law $P \parallel R = Q \parallel R$ is not satisfied.

[3] $tr_i$ returns the $(i+1)th$ element of the sequence $tr$.

## 6 Conclusion

In this work, we have proposed an observation-oriented semantics in an open environment 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 have been proposed based on the denotational model for communicating processes involving shared variables. Furthermore, a *closed* semantics has been derived from the open denotational semantics by focusing on the particular hybrid traces. Our next step is to encode our proposed semantics into a generic theorem prover and in turn to validate the algebraic laws using the theorem proving techniques. Ultimately, we can verify the correctness of system behaviours in a theorem prover which may solve the common state space explosion problem.

**Acknowledgements.** The authors thank Jim Woodcock for insightful comments in the initial discussion.

## References

1. Brookes, S.D.: Full abstraction for a shared-variable parallel language. Inf. Comput. 127(2), 145–163 (1996)
2. Cavalcanti, A., Woodcock, J.: A tutorial introduction to CSP in *unifying theories of programming*. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 220–268. Springer, Heidelberg (2006)
3. Colvin, R., Hayes, I.J.: CSP with Hierarchical State. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 118–135. Springer, Heidelberg (2009)
4. Fischer, C.: Combining Object-Z and CSP. In: FBT, pp. 119–128 (1997)
5. Galloway, A.J., Stoddart, W.J.: An Operational Semantics for ZCCS. In: ICFEM, pp. 272–282 (1997)
6. Hoare, C.: Communicating Sequential Processes. Prentice-Hall (1985)
7. Hoare, C., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
8. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP Semantics for *Circus*. Formal Asp. Comput. 21(1-2), 3–32 (2009)
9. Qin, S., Dong, J.S., Chin, W.-N.: A Semantic Foundation for TCOZ in Unifying Theories of Programming. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 321–340. Springer, Heidelberg (2003)
10. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1997)
11. Schneider, S., Treharne, H.: CSP Theorems for Communicating B Machines. Formal Asp. Comput. 17(4), 390–422 (2005)
12. Shi, L.: A UTP Semantics for Communicating Processes with Shared Variables. Technical report, NUS (2013), `http://www.comp.nus.edu.sg/~shiling/Tech13.pdf`
13. Smith, G.: A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In: Fitzgerald, J.S., Jones, C.B., Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 62–81. Springer, Heidelberg (1997)
14. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating Specification and Programs for System Modeling and Verification. In: TASE, pp. 127–135 (2009)
15. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
16. Taguchi, K., Araki, K.: The State-Based CCS Semantics for Concurrent Z Specification. In: ICFEM, pp. 283–292 (1997)

17. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)
18. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal Methods: Practice and Experience. ACM Comput. Surv. 41(4) (2009)
19. Huibiao, Z., Bowen, J.P., Jifeng, H.: From Operational Semantics to Denotational Semantics for Verilog. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 449–464. Springer, Heidelberg (2001)
20. Zhu, H., He, J., Bowen, J.P.: From algebraic semantics to denotational semantics for verilog. In: ISSE, vol. 4(4), pp. 341–360 (2008)
21. Zhu, H., Qin, S., He, J., Bowen, J.P.: PTSC: probability, time and shared-variable concurrency. In: ISSE, vol. 5(4), pp. 271–284 (2009)
22. Zhu, H., Yang, F., He, J., Bowen, J.P., Sanders, J.W., Qin, S.: Linking Operational Semantics and Algebraic Semantics for a Probabilistic Timed Shared-Variable Language. J. Log. Algebr. Program. 81(1), 2–25 (2012)

# Appendix

## A Proof of Lemma 1

**Proof.** We show here the proof of monotonicity for the parallel composition operator, and the proofs of monotonicity for other operators are available in our technical report [12].

**Case: Parallel Composition** Given any two processes $P$ and $Q$ such that $P \sqsupseteq Q$ and synchronisation events of process $P \parallel R$ and process $Q \parallel R$ are the same (denoted as set $X$), $P \parallel R \sqsupseteq Q \parallel R$ holds.

First, we have two auxiliary lemmas in our proof, whose proofs are available in [12].

**Lemma 2.** *$(P \wedge R) \sqsupseteq (Q \wedge R)$, provided that $P \sqsupseteq Q$.*

**Lemma 3.** *$(P;\ R) \sqsupseteq (Q;\ R)$, provided that $P \sqsupseteq Q$.*

$$
\begin{array}{lll}
 & P \sqsupseteq Q & [\textit{Definition } 2] \\
= & [P \Rightarrow Q] & [\textit{predicate calculus}] \\
= & [P[0.obs/obs'] \Rightarrow Q[0.obs/obs']] & [\textit{Definition } 2] \\
= & P[0.obs/obs'] \sqsupseteq Q[0.obs/obs'] & [\textit{Lemma } 2] \\
= & \begin{array}{l}(P[0.obs/obs'] \wedge R[1.obs/obs']) \ \sqsupseteq \\ (Q[0.obs/obs'] \wedge R[1.obs/obs'])\end{array} & [\textit{Lemma } 3] \\
\Rightarrow & \begin{array}{l}(P[0.obs/obs'] \wedge R[1.obs/obs']);\ M(X) \ \sqsupseteq \\ (Q[0.obs/obs'] \wedge R[1.obs/obs']);\ M(X)\end{array} & [3.2.3] \\
= & P \parallel R \sqsupseteq Q \parallel R & \square
\end{array}
$$

In a similar proof, the predicate $R \parallel P \sqsupseteq R \parallel Q$ also holds given the same assumption as above. □

## B Proof of Theorem 1

**Proof.** Given 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 3. According to Lemma 1, both $F(P) \sqsupseteq F(Q)$ and $F(Q) \sqsupseteq F(P)$, which indicates $F(P) = F(Q)$, i.e., the open semantics is compositional. □