Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 250 (2009) 103–118

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

PVS Embedding of cCSP Semantic Models and Their Relationship

Shamim H Ripon[1]

Department of Computing Science, University of Glasgow, Glasgow, Scotland

Michael J Butler[2]

School of Electronics and Computer Science, University of Southampton, Southampton, UK

Abstract

This paper demonstrates an embedding of the semantic models of the cCSP process algebra in the general purpose theorem prover PVS. cCSP is a language designed to model long-running business transactions with constructs for orchestration of compensations. The cCSP process algebra terms are defined in PVS by using mutually recursive datatype. The trace and the operational semantics of the algebra are embedded in PVS. We show how these semantic embeddings are used to define and prove a relationship between the semantic models by using the powerful induction mechanism of PVS.

Keywords: Compensating CSP, semantic models, embedding, induction, PVS.

1 Introduction

Compensating CSP (cCSP) [5] is a language designed to model long-running business transactions within the framework of standard CSP process algebra [13]. Business transactions typically involve multiple partners coordinating and interacting with each other. *Compensation* is defined in [11] as an action taken to recover from error in business transactions, particularly, in long-running transactions. cCSP provides constructs for orchestration of compensations to model business transactions.

A formal semantics offers a complete, and rigorous definition of a language. Operational and denotational semantics are two well-known methods of assigning meaning to languages and both semantics are useful for a full description of a language. Traces are one of the ways to define denotational semantics. A trace

[1] Email: shamim@dcs.gla.ac.uk

[2] Email: mjb@ecs.soton.ac.uk

doi:10.1016/j.entcs.2009.08.020

gives the global picture of the behaviour of a system. The trace semantics of cCSP is defined in [5]. Operational semantics describes the behaviour of programs in terms of the transitions between program states, or configurations. The operational semantics of cCSP is defined in [6] by using labelled transition systems [17]. Both semantics have valuable applications and a key question is *"How are they related?"*. A relationship between the semantic models has been defined, and proved, where the proofs have been carried out completely by hand [20]. In this paper, we investigate how a theorem prover can support the mechanical verification of those hand proofs.

Proof by hand is a difficult and tedious task to manage a large number of steps. Subtle mistakes, or omissions can easily occur at any stage of such proofs. A tool that allows mechanising the semantic models, and supports mechanically proving the relationship between the semantic models can overcome the problems in the hand proofs. An interactive and automatic theorem prover that successfully mechanise our proofs, demonstrates the correctness of our proofs, and a feasible mechanisation allows us to follow the same mechanical proof technique to apply to larger proofs. To address our problem, presently, there are several systems, such as, PVS [14], HOL [10], Isabelle [16], and Coq [3], having a rich specification language and automated support for decision procedures, and proof strategies to their logic.

PVS is an automated framework for specification and verification. PVS supports high-order logic, allows abstract datatypes to model process terms, and has a strong support for induction mechanism. PVS supports interactive proof checking, where the user applies proof commands to simplify the goal to be proved, until it can be proved automatically by its decision procedure. Several research, such as [4,8,9], have been carried out in order to mechanise process algebras by using PVS, where an ACP-style process algebra [2] has been mechanised in [4], and the trace semantics of standard CSP has been mechanised in [8,9]. The cCSP semantic models are closely related to that of standard CSP, and given the positive experience described in [8,9], we have decided to use PVS in our experiments. Most of the existing works in PVS are aimed at concrete applications, and very few of them are focused in the theoretical issues, especially, process algebra terms and semantic models. To the best of our knowledge, it is the first attempt to mechanise both the operational, and the trace models of an algebra as well as their relationship in PVS.

After giving a brief overview of the cCSP language, we outline the definitions of the traces, and the operational semantics. We briefly describe how a relationship is defined between the two semantic models, and how they are proved by hand. We then describe the embedding of cCSP in PVS. The process algebra terms are defined by using a mutually recursive datatype. The traces are defined by following the original semantic definitions. A recursive definition is given to define the operational semantics that plays a vital role in the proofs. We define several supporting lemmas and mechanise them to prove the relationship between the semantic models. The lemmas are proved by applying induction and the proof steps follow similar level of granularity as in the hand proofs.

2 Compensating CSP

The introduction of the cCSP language was inspired by the combination of two ideas: transaction processing features, and process algebra. Like standard CSP, processes in cCSP are modelled in terms of the atomic events they can engage in. The language provides operators that support sequencing, choice, parallel composition of processes. In order to support failed transaction, compensation operators are introduced. The processes are categorised into standard, and compensable processes. A standard process does not have any compensation, but compensation is part of a compensable process that is used to compensate a failed transaction. We use notations, such as, $P, Q, ..$ to identify standard processes, and $PP, QQ, ..$ to identify compensable processes. A subset of the original cCSP is considered in this paper, which includes most of the operators. The cCSP syntax, considered in this paper, is summarised in Fig. 1.

Standard Processes:			**Compensable Processes:**		
$P, Q ::=$	A	(atomic event)	$PP, QQ ::=$	$P \div Q$	(compensation pair)
	$\mid P \,;\, Q$	(sequential composition)		$\mid PP \,;\, QQ$	
	$\mid P \,\Box\, Q$	(choice)		$\mid PP \,\Box\, QQ$	
	$\mid P \parallel Q$	(parallel composition)		$\mid PP \parallel QQ$	
	$\mid \mathit{SKIP}$	(normal termination)		$\mid \mathit{SKIPP}$	
	$\mid \mathit{THROW}$	(throw an interrupt)		$\mid \mathit{THROWW}$	
	$\mid \mathit{YIELD}$	(yield to an interrupt)		$\mid \mathit{YIELDD}$	
	$\mid P \rhd Q$	(interrupt handler)			
	$\mid [PP]$	(transaction block)			

Fig. 1. cCSP syntax

The basic unit of the standard processes is an atomic event (A). The other operators are the sequential ($P \,;\, Q$), and the parallel composition ($P \parallel Q$), the choice operator ($P \,\Box\, Q$), the interrupt handler ($P \rhd Q$), the empty process SKIP, raising an interrupt THROW, and yielding an interrupt YIELD. A process that is ready to terminate is also willing to yield to an interrupt. In a parallel composition, throwing an interrupt by one process synchronises with yielding in another process. Yield points are inserted in a process through YIELD. For example, ($P \,;\, \mathit{YIELD} \,;\, Q$), is willing to yield to an interrupt in between the execution of P, and Q. The basic way of constructing a compensable process is through a compensation pair ($P \div Q$), which is constructed from two standard processes, where P is called the forward behaviour that executes during normal execution, and Q is called the associated compensation that is designed to compensate the effect of P when needed. The sequential composition of compensable processes is defined in such a way that the compensations of the completed tasks will be accumulated in reverse to the order of their original composition, whereas compensations from the compensable parallel processes will be placed in parallel. In this paper, we define only the asynchronous composition of processes, where processes interleave with each other during normal execution, and synchronise during termination. By enclosing a compensable process PP inside a transaction block $[\,PP\,]$, we get a complete transaction and the transaction block itself is a standard process. Successful completion of PP represents successful completion of the block. But, when the forward behaviour of PP throws an interrupt, the compensations are executed inside the block, and the interrupt is not observable from outside of the block. *SKIPP*, *THROWW*, and *YIELDD* are the compensable counterpart of the corresponding standard processes and they are defined as follows:

$$SKIPP = SKIP \div SKIP, \qquad YIELDD = YIELD \div SKIP$$
$$THROWW = THROW \div SKIP$$

2.1 Trace Semantics

A trace records the history of behaviour of a process up to some point. In the cCSP trace model, only completed traces are considered for which a process terminates. This simplifies many definitions since the nature of a trace is indicated by its final symbol. A trace is defined as a sequence of events (non-terminal) followed by a terminal event, written as $s\langle\omega\rangle$, where $s \in \Sigma^*$, where Σ is the alphabet of non-terminal events, and $\omega \in \Omega$, where Ω is the set of terminal events ($\Omega = \{\checkmark, !, ?\}$). So, a trace can be $s\langle\checkmark\rangle$ (successful termination), or $s\langle\,!\,\rangle$ (terminates throwing an interrupt), or $s\langle ? \rangle$ (yield to an interrupt). Processes denote non-empty sets of such traces. For traces s and t, we write $s.t$ as their concatenation. Operators are first defined on traces and then lifted to sets of traces to define processes. A compensable process has forward and compensation behaviour, and is modelled as a pair of traces of the form $(s\langle\omega\rangle, s'\langle\omega'\rangle)$, where $s\langle\omega\rangle$ represents the forward behaviour, and $s'\langle\omega'\rangle$ represents the compensation. For processes P and PP, we write $T(P)$, and $T(PP)$ respectively, to denote their traces. The trace semantics of some cCSP operators are outlined in Fig. 2 (refer to [5] for details). If ω and ω' are terminal events from distinct parallel processes, the joint terminal event of their parallel execution is denoted by $\omega \,\&\, \omega'$ (Fig. 2). The synchronisation operator is defined to be commutative and case analysis shows that it is associative.

Atomic Action:	$T(A) = \{\langle A, \checkmark\rangle\}, A \in \Sigma$
Sequential Composition:	$p\langle\checkmark\rangle \; ; \; q = p.q$, and $p\langle\omega\rangle \; ; \; q = p\langle\omega\rangle$, where $\omega \neq \checkmark$
	$T(P \; ; \; Q) = \{p \; ; \; q \mid p \in T(P) \wedge q \in T(Q)\}$
Parallel Composition:	$p\langle\omega\rangle \| q\langle\omega'\rangle = \{r\langle\omega\&\omega'\rangle \mid r \in (p \,\|\|\|\, q)\}$ where $\begin{array}{c\|cccccc} \omega & ! & ! & ! & ? & ? & \checkmark \\ \omega' & ! & ? & \checkmark & ? & \checkmark & \checkmark \\ \hline \omega\&\omega' & ! & ! & ! & ? & ? & \checkmark \end{array}$
	$T(P \| Q) = \{r \mid r \in (p \| q) \wedge p \in T(P) \wedge q \in T(Q)\}$
Compensation Pair:	$p\langle\checkmark\rangle \div q = (p\langle\checkmark\rangle, q)$ and $p\langle\omega\rangle \div q = (p\langle\omega\rangle, \langle\checkmark\rangle)$ where $\omega \neq \checkmark$
	$T(P \div Q) = \{p \div q \mid p \in T(P) \wedge q \in T(Q)\}$
Transaction Block:	$[p\langle !\rangle, p'] = p.p'$ and $[p\langle\checkmark\rangle, p'] = p\langle\checkmark\rangle$
	$T([PP]) = \{[p, p'] \mid (p, p') \in T(PP)\}$

Fig. 2. A part of cCSP trace semantics

If PP contains only yielding behaviour then $[\,PP\,]$ would be empty. The following healthiness conditions ensure that $[\,PP\,]$ is non-empty by declaring that all processes P and PP consist of some terminating, or interrupting behaviour:

- $p\langle\checkmark\rangle \in T(P)$ or $p\langle\,!\,\rangle \in T(P)$, for some p
- $(p\langle\checkmark\rangle, p') \in T(PP)$ or $(p\langle\,!\,\rangle, p') \in T(PP)$, for some p, p'

2.2 Operational Semantics

The operational semantics defines the transitions of process terms from one state to another. Transition rules are defined for both standard and compensable processes. For each process term, there are two types of transition rules: normal transition (by a normal event), and terminal transition (by a terminal event). A normal transition takes a process term from one state to its another state, and a terminal transition causes the termination of a process term. The language terms are extended with a null process (0) to denote the termination of a standard process. For example, a normal event a makes the transition of process terms P and PP, to their respective another state P' and PP', whereas a terminal event ω makes the standard process terminate to a null process, and terminates the forward behaviour of the compensable process leaving its compensation:

$$P \xrightarrow{a} P' \ (P' \text{ is a standard process}) \qquad PP \xrightarrow{a} PP' \ (PP' \text{ is a compensable process})$$
$$P \xrightarrow{\omega} 0 \qquad\qquad PP \xrightarrow{\omega} P \quad (P \text{ is the compensation})$$

Assuming $a \in \Sigma$, and $\omega \in \Omega$, the transition rules for some cCSP operators are summarised in Fig. 3. For a detailed discussion please refer to [6].

Atomic action: $A \xrightarrow{A} \mathit{SKIP} \quad (A \in \Sigma)$

Sequential composition: $\dfrac{P \xrightarrow{a} P'}{(P\ ;\ Q) \xrightarrow{a} (P'\ ;\ Q)} \quad \dfrac{P \xrightarrow{\checkmark} 0 \ \wedge\ Q \xrightarrow{\alpha} Q'}{(P\ ;\ Q) \xrightarrow{\alpha} Q'}\ (\alpha \in \Sigma \cup \Omega) \quad \dfrac{P \xrightarrow{\omega} 0}{(P\ ;\ Q) \xrightarrow{\omega} 0}\ (\omega \neq \checkmark)$

Parallel composition: $\dfrac{P \xrightarrow{a} P'}{(P \parallel Q) \xrightarrow{a} (P' \parallel Q)} \quad \dfrac{Q \xrightarrow{a} Q'}{(P \parallel Q) \xrightarrow{a} (P \parallel Q')} \quad \dfrac{P \xrightarrow{\omega 1} 0 \ \wedge\ Q \xrightarrow{\omega 2} 0}{(P \parallel Q) \xrightarrow{\omega 1 \& \omega 2} 0}\ (\omega 1, \omega 2 \in \Omega)$

Compensation Pair: $\dfrac{P \xrightarrow{a} P'}{(P \div Q) \xrightarrow{a} (P' \div Q)} \quad \dfrac{P \xrightarrow{\checkmark} 0}{(P \div Q) \xrightarrow{\checkmark} Q} \quad \dfrac{P \xrightarrow{\omega} 0}{(P \div Q) \xrightarrow{\omega} \mathit{SKIP}}\ (\omega \neq \checkmark)$

Transaction Block: $\dfrac{PP \xrightarrow{a} PP'}{[PP] \xrightarrow{a} [PP']} \quad \dfrac{PP \xrightarrow{\checkmark} P}{[PP] \xrightarrow{\checkmark} 0} \quad \dfrac{PP \xrightarrow{!} P \ \wedge\ P \xrightarrow{\alpha} P'}{[PP] \xrightarrow{\alpha} P'}\ (\alpha \in \Sigma \cup \Omega)$

Fig. 3. A part of cCSP operational semantics

Note that there is no transition rule for a yield (?) in a transition block, because a transaction block does not yield to an interrupt from outside. Yield by a subprocess of PP will synchronise with the interrupt thrown by some other sub-process, resulting the ! event, making yield within PP non-observable.

3 Relationship Between the Semantic Models

We have adopted a systematic approach to derive a relationship between the semantic models. First, traces are extracted (derived trace) from the transition rules of the operational semantics. Then, a correspondence is established between the extracted traces with the originally defined traces.

The operational semantics leads to the lifted transition relation labelled by sequences of events. This is defined recursively. For a standard process P:

$$P \xrightarrow{\langle\omega\rangle} Q \;=\; P \xrightarrow{\omega} Q$$
$$P \xrightarrow{\langle a\rangle t} Q \;=\; \exists\, P' \cdot P \xrightarrow{a} P' \,\wedge\, P' \xrightarrow{t} Q$$

For a standard process P, the derived trace, $DT(P)$, is defined as follows:

Definition 3.1 For a standard trace t, $\quad t \in DT(P) \;=\; P \xrightarrow{t} 0$

The derived trace t consists of a sequence of events followed by a terminal event. For a compensable process PP, and its pair of traces t and t', we define that,

$$PP \xrightarrow{(t,t')} 0 \;=\; \exists\, R \cdot PP \xrightarrow{t} R \,\wedge\, R \xrightarrow{t'} 0 \quad (R \text{ is the attached compensation})$$

The derived traces of a compensable process is then defined as follows:

Definition 3.2 For traces t and t', $\quad (t,t') \in DT(PP) \;=\; PP \xrightarrow{(t,t')} 0$

We state the following theorem to show that the derived traces correspond to the originally defined traces:

Theorem 3.3 *For a standard process P, where $P \neq 0$, $\quad DT(P) \;=\; T(P)$. For a compensable process PP, where $PP \neq 0$, $\quad DT(PP) \;=\; T(PP)$.*

The theorem can be proved by using structural induction over process terms. We outline here how we define some lemmas that support the proof of the structural cases.

Traces are extracted for each term of the language and show their correspondence with the original trace semantics. For standard processes, P and Q, for all the operators, we show that,

$$t \in DT(P \otimes Q) \;=\; t \in T(P \otimes Q) \tag{1}$$

For each such operator $\otimes$, the proof is performed by induction over traces, and it is carried out by assuming that, $DT(P) = T(P)$, and $DT(Q) = T(Q)$. Similarly, for compensable processes, PP and QQ, we show that,

$$(t,t') \in DT(PP \otimes QQ) \;=\; (t,t') \in T(PP \otimes QQ) \tag{2}$$

Consider the sequential composition of processes P and Q. By using (1), the relationship between the semantic models is derived by showing that,

$$t \in DT(P \,;\, Q) \;=\; t \in T(P \,;\, Q)$$

From Def. 3.1, we get the derived traces of the sequential composition,

$$t \in DT(P \,;\, Q) \;=\; (P \,;\, Q) \xrightarrow{t} 0$$

We also expand the definition of trace semantics as follows:

$$\begin{array}{ll}
t \in\; T(P \,;\, Q) & \\
\quad = \exists\, p, q \cdot t = (p \,;\, q) \;\wedge\; p \in T(P) \;\wedge\; q \in T(Q) & \text{[Trace definition]} \\
\quad = \exists\, p, q \cdot t = (p \,;\, q) \;\wedge\; p \in DT(P) \;\wedge\; q \in DT(Q) & \text{[Induction assumption]} \\
\quad = \exists\, p, q \cdot t = (p \,;\, q) \;\wedge\; P \xrightarrow{p} 0 \;\wedge\; Q \xrightarrow{q} 0 & \text{[Derived trace definition]}
\end{array}$$

Finally, from the above definitions of traces, the following lemma is formulated for the sequential composition of standard processes:

Lemma 3.4 $(P\ ;\ Q) \xrightarrow{t} 0\ =\ \exists\, p, q \cdot t = (p\ ;\ q)\ \wedge\ P \xrightarrow{p} 0\ \wedge\ Q \xrightarrow{q} 0$

The lemma is proved by applying induction over the trace t, where $t = \langle\omega\rangle$ is considered as the base case, and $t = \langle a\rangle t$ is considered as the inductive case. In order to support the inductive proof, two supporting equations are derived from the transition rules, based on the types of events by which the transition rules are defined:

$$(P\ ;\ Q) \xrightarrow{\omega} 0\ =\ \ P \xrightarrow{\checkmark} 0 \wedge Q \xrightarrow{\omega} 0\quad \vee\quad P \xrightarrow{\omega} 0 \wedge \omega \neq \checkmark \tag{3}$$

$$(P\ ;\ Q) \xrightarrow{a} R\ =\ \ \exists\, P' \cdot P \xrightarrow{a} P'\ \wedge\ R = (P'\ ;\ Q)\quad \vee\quad P \xrightarrow{\checkmark} 0\ \wedge\ Q \xrightarrow{a} R \tag{4}$$

Following similar steps, the lemma for the parallel composition is defined as follows:

Lemma 3.5 $(P \parallel Q) \xrightarrow{t} 0\ =\ \exists\, p, q \cdot t \in (p \parallel q)\ \wedge\ P \xrightarrow{p} 0\ \wedge\ Q \xrightarrow{q} 0$

For compensable processes, it is only required to derive traces for the forward behaviour, and reuse the derived traces from the standard processes for the compensations. For example, the lemma for the lifted forward behaviour of the parallel composition of compensable processes is defined as follows:

Lemma 3.6 $(PP \parallel QQ) \xrightarrow{t} R\ =$
$\exists\, P, Q, p, q \cdot t \in (p \parallel q)\ \wedge\ PP \xrightarrow{p} P\ \wedge\ QQ \xrightarrow{q} Q\ \wedge\ R = (P \parallel Q)$

Both the compensation pair, and the transaction block have special behaviour. The compensable behaviour of a compensation pair $(P \div Q)$, is defined by the standard behaviour of P, and Q. On the other hand, the standard behaviour of a transaction block $[\,PP\,]$, is defined by the behaviour of the compensable process PP. Lemmas for these two operators are defined as follows:

Lemma 3.7 $(P \div Q) \xrightarrow{(t,t')} 0\ =\ \exists\, p, q \cdot (t, t') = (p \div q)\ \wedge\ P \xrightarrow{p} 0\ \wedge\ Q \xrightarrow{q} 0$

Lemma 3.8 $[\,PP\,] \xrightarrow{t} 0\ =\ \exists\, p, p' \cdot t = [\,p, p'\,]\ \wedge\ PP \xrightarrow{p,p'} 0$

In this section, we have only outlined how the lemmas are defined, and how they can be proved. A detailed discussion of the hand proofs of all the lemmas can be found in [19,20].

4 PVS Mechanisation

A way to combine the strength of general purpose theorem provers with formal notations is the semantic embedding of the formal notations within the logic of the verification systems. An embedding is a semantic encoding of one specification language into another, especially, to reuse the existing tools of the target language. There are two main variants of semantic embedding: *deep* and *shallow* embedding [18]. In a deep embedding, the language and the semantics of the method are fully formalised as an object in the logic of the specification language. On the other hand, in a shallow embedding, there is a syntactic translation of the objects of the method into semantically equivalent objects of the verification system. Shallow embedding concentrates on the semantic embedding of the guest logic into the host logic, and it is easy to set up. For our purpose, we use shallow embedding to define cCSP in PVS.

4.1 cCSP Syntax

First, we define the process terms to define the cCSP syntax. Separate syntax is used to define the standard, and the compensable processes. As PVS supports overloading, same notations can be used for the operational and the trace semantics. Fig. 4 summarises the PVS definition of cCSP syntax.

Standard			**Compensable**		
	PVS			PVS	
cCSP	(Operational)	(Trace)	**cCSP**	(Operational)	(Trace)
A	`act(a)`	`act(a)`			
$SKIP$	`Skip`	`SKIP`	$SKIPP$	`Skipp`	`SKIPP`
$THROW$	`Throw`	`THROW`	$THROWW$	`Throww`	`THROWW`
$YIELD$	`Yield`	`YIELD`	$YIELDD$	`Yieldd`	`YIELDD`
$P \Box Q$	`choice(P,Q)`	`choice(P,Q)`	$PP \Box QQ$	`cchoice(PP,QQ)`	`cchoice(PP,QQ)`
$P \; ; \; Q$	`seq(P,Q)`	`seq(P,Q)`	$PP \; ; \; QQ$	`cseq(PP,QQ)`	`cseq(PP,QQ)`
$P \parallel Q$	`para(P,Q)`	`parallel(P,Q)`	$PP \parallel QQ$	`cpara(PP,QQ)`	`parallel(PP,QQ)`
$P \rhd Q$	`P |> Q`	`intr(P,Q)`	$P \div Q$	`cpair(P,Q)`	`cpair(P,Q)`
$[PP]$	`blk(PP)`	`block(PP)`			

Fig. 4. cCSP syntax in PVS

4.2 Events, Traces and Processes

There are two types of events defined in cCSP: observable (normal), and terminal. These two types are defined in PVS, and three terminal events are defined as constants of terminal type as follows:

```
normal  : TYPE
terminal: TYPE+
skip, yield, throw : terminal
```

The keyword `TYPE+` indicates that `terminal` is a *non-empty* type, that allows to define constants of that type.

The traces of cCSP are defined in PVS by following the original trace definition, as a pair consisting of a list of normal event, and a terminal event. The definition ensures that traces are non-empty (at least there is a terminal event when the list is empty). Compensable traces are defined as pair of standard traces. Processes are defined as a set of traces.

```
trace         : TYPE = [list[normal],terminal]
comp_trace    : TYPE = [trace, trace]
process       : TYPE = setof[trace]
```

```
comp_process : TYPE = setof[comp_trace]
```

4.3 Process Algebra Terms

Proofs about properties of a process algebra often use induction on the structure of the algebra, which is no exception in our case. PVS has a datatype called abstract datatype, for which PVS generated an induction scheme, and it is convenient to model process algebra terms as an abstract datatype. PVS provides mechanism to define abstract datatype of certain class, which includes all of the tree-like recursive data structure that are freely generated by a number of constructor operators (detailed discussion in [15]).

cCSP has standard, and compensable process terms and importantly, these process terms are mutually dependant on each other. We have already mentioned the mutual dependency of the compensation pair, and the transaction block. Hence, to model cCSP process terms we need a support to define mutually recursive datatype; but, mutually recursive datatype is not directly admissible by PVS. However, PVS has an extended support of *sub-datatype* [15,21], where it is possible to define two mutually recursive datatypes as a single datatype. A sub-datatype collects together groups of constructors of a datatype that form one part of a mutually recursive datatype definition. Taking this facility, the cCSP process algebra terms with two sub-datatypes are defined in Fig. 5.

```
  pa_terms  : DATATYPE WITH SUBTYPES stand, comp
    BEGIN
       Skip    : skip?   : stand
       Throw   : throw?  : stand
       Yield   : yield?  : stand
       act(a:normal)                   : act?      : stand
       choice(P: stand, Q: stand)      : choice?   : stand
       seq(P:stand, Q:stand)           : seq?      : stand
       para(P:stand, Q:stand)          : para?     : stand
       |>(P: stand, Q: stand)          : inthnd?   : stand
       nul                             : nul?      : stand
       cseq(PP : comp, QQ : comp)      : c_seq?    : comp
       cpara(PP : comp, QQ :comp)      : c_para?   : comp
       cchoice(PP : comp, QQ : comp)   : c_choice? : comp
       cpair(P: stand, Q : stand)      : cpair?    : comp
       blk(PP : comp)                  : blk?      : stand
 END pa_terms
```

Fig. 5. Process algebra terms in PVS

We define a single datatype `pa_terms` that consists of two sub-datatypes: '`stand`' for standard processes, and '`comp`' for compensable processes. We can now define process terms of types '`stand`' and '`comp`'. We define the additional process term '`nul`' to denote the null (0) process that has been used in the definitions of the operational semantics. The compensable basic processes can be defined by using the already defined datatype.

5 Mechanising the Trace Semantics

The trace semantics are defined in PVS in the same way as they are originally defined. Operators are first defined at the trace level, and then lift to the sets of traces to define the processes. The same approach is taken for both standard, and compensable processes. Within the limited scope in this paper, we outline the trace definitions of only a few operators. Other operators are defined by following a similar approach.

For standard traces p and q, their sequential composition $(p\ ;\ q)$, is defined in such a way that if p ends with a $\checkmark$, the trace q will be augmented to the observable behaviour of the trace p, and $\checkmark$ will be hidden from the environment. When p ends with a terminal event other than $\checkmark$, q is discarded. This definition is then lifted to define the sequential composition of standard processes. The PVS definition is given as follows:

```
 seq(p,q : trace) : trace =
    if PROJ_2(p) = tick THEN
      (append(PROJ_1(p),PROJ_1(q)), PROJ_2(q))
    ELSE p   ENDIF
seq(P,Q:process):process = {t:trace|EXISTS (p:(P),q:(Q)):t=seq(p,q)}
```

`PROJ_i` represents the *ith*[3] element of a tuple. The parallel composition of processes is defined as the interleaving of the observable events followed by the synchronisation of the terminal events. The interleaving of observable events is defined as follows:

```
 interleave(t1,t2,t:list[normal]): RECURSIVE bool =
 CASES t OF
      null: null?(t1) AND null?(t2),
 cons(x,y): (cons?(t1) AND car(t1)= x AND interleave(cdr(t1),t2,y))
         OR (cons?(t2) AND car(t2)= x AND interleave(t1,cdr(t2),y))
 ENDCASES  MEASURE length(t)
```

`interleave(t1,t2,t)` holds when `t` is a valid interleaving of `t1` and `t2`. PVS allows a restrictive form of recursive definition. Mutual recursion is not allowed, and the function must be total. In order to ensure this, a `MEASURE` function is required, which is defined to show that the definition terminates by generating an obligation that `MEASURE` decreases with each call. `MEASURE` can range over `nat` (natural number), or ordinals. Following the definition of interleaving, the synchronisation of terminal event is defined as follows:

```
   parallel(w:terminal)(w1,w2:terminal): bool =
    IF w = throw THEN w1 = throw AND w2 = throw
         OR w1 = throw AND w2 = yield OR w1 = throw AND w2 = tick
         OR w1 = yield AND w2 = throw OR w1 = tick  AND w2 = throw
    ELSIF w = yield THEN w1 = yield AND w2 = yield
         OR w1 = yield AND w2 = tick OR w1 = tick  AND w2 = yield
```

[3] the *ith* element of a tuple `t` can also be presented as `t‘i`.

```
    ELSE  w1 = tick  AND w2 = tick
    ENDIF
```

Finally, the parallel composition of traces is defined by combining the interleaving of list of events, and the synchronisation of the terminal events as follows:

```
 parallel(r:trace)(p,q: trace): bool =
   interleave(proj_1(p),proj_1(q),proj_1(r))
   AND parallel(proj_2(r))(proj_2(p),proj_2(q))
 parallel(P,Q: process): process =
     {t:trace | EXISTS (p:(P),q:(Q)): parallel(t)(p,q)}
```

Compensable processes are defined by following similar approach. For compensable processes, we need the additional definition for the compensations, and they are same as that of standard processes. The order of installation of the compensations depend on the operators. For example, compensations from the sequential composition are installed in reverse to their original order, whereas they are installed in parallel for the parallel composition. We also define here the trace semantics for both the compensation pair, and the transaction block as follows:

```
pair(p,q:trace): comp_trace =
    IF p`2 = tick THEN (p,q) ELSE (p,(null,tick)) ENDIF
pair(P,Q:process): comp_process =
   {tt: comp_trace | EXISTS (p:(P),q:(Q)): tt = pair(p,q)}
block(pp:comp_trace) : trace =
 IF (pp`1)`2 = throw THEN (append((pp`1)`1,(pp`2)`1),(pp`2)`2)
    ELSE  pp`1 ENDIF
block(PP:comp_process): process =
   { t: trace | EXISTS (pp:(PP)): t = block(pp) }
```

6 Mechanising the Operational Semantics

The operational semantics is defined by using labelled transition systems of the form $P \xrightarrow{e} P'$, where the event e makes the transition of the process term from state P to P'. Two types of transitions are defined: normal, and terminal. Both transition rules are defined by using a recursive boolean definition that determines whether there is a transition from one state to another state. The definitions are given by using the derived equations from the transition rules, which are used in the proofs of the lemmas for the process terms (e.g., equation (3) and (4) are used for sequential composition). The transition rules of some process terms depend on the transition rules of both standard and compensable processes. To define these rules, we need to combine the transition rules for both standard and compensable processes. For normal transitions, this can be done easily. But, care has to be taken for terminal transitions because the terminal transitions of the standard, and the compensable processes are different. We have defined the '`nul`' as a standard process, which has allowed to combine the terminal transitions of both process terms. The terminal transitions of a few process terms are shown in Fig. 6.

```
wtrans(w: terminal)(P:pa_terms,P1:stand): RECURSIVE bool =
 CASES P OF
  ..
 seq(Q,R)     : wtrans(tick)(Q,nul) AND wtrans(w)(R,nul)
           OR  w /= tick AND wtrans(w)(Q,nul),
 para(Q,R)    : EXISTS (w1,w2:terminal): wtrans(w1)(Q,nul)
               AND wtrans(w2)(R,nul) AND parallel(w)(w1,w2),
 cseq(QQ,RR) : EXISTS (Q,R : stand):
               wtrans(tick)(QQ,Q) AND wtrans(w)(RR,R) AND P1 = seq(R,Q)
           OR  wtrans(w)(QQ,P1) AND w /= tick,
 cpara(QQ,RR): EXISTS (w1,w2:terminal, Q,R:stand):
               wtrans(w1)(QQ,Q) AND wtrans(w2)(RR,R) AND
               parallel(w)(w1,w2) AND P1 = para(Q,R),
 cpair(Q,R)  : wtrans(tick)(Q,nul) AND P1 = R
           OR  wtrans(w)(Q,nul) AND w /= tick AND P1 = Skip,
 blk(QQ)     : (EXISTS (Q:stand): wtrans(throw)(QQ,Q) AND wtrans(w)(Q,nul))
           OR  (EXISTS (Q:stand): w = tick AND wtrans(w)(QQ,Q)),
   ..
 ENDCASES
 MEASURE P BY <<
```

Fig. 6. Terminal transitions in PVS

The normal transitions for both standard and compensable processes are also defined together. A part of the definition is given in Fig. 7. The `MEASURE` keyword

```
ntrans(a:normal)(Pa: pa_terms, Pa1: pa_terms): RECURSIVE bool =
 CASES Pa OF
  ..
seq(Q,R)     : EXISTS (Q1:stand): ntrans(a)(Q,Q1) AND Pa1 = seq(Q1,R)
          OR  wtrans(tick)(Q,nul) AND ntrans(a)(R,Pa1),
para(Q,R)    : EXISTS (Q1:stand): ntrans(a)(Q,Q1) AND Pa1 = para(Q1,R)
          OR  EXISTS (R1:stand): ntrans(a)(R,R1) AND Pa1 = para(Q,R1),
cseq(QQ,RR) : EXISTS (QQ1:comp): ntrans(a)(QQ,QQ1) AND Pa1 = cseq(QQ1,RR)
          OR  EXISTS (RR1:comp, Q:stand): wtrans (tick)(QQ,Q) AND
              ntrans(a)(RR,RR1) AND Pa1 = ax(RR1,Q),
cpara(QQ,RR): EXISTS (QQ1:comp): ntrans(a)(QQ,QQ1)  AND Pa1 = para(QQ1,RR)
          OR  EXISTS (RR1:comp): ntrans(a)(RR,RR1)  AND Pa1 = para(QQ,RR1),
cpair(Q,R)  : EXISTS (Q1:stand): ntrans(a)(Q,Q1)    AND Pa1 = cpair(Q1,R),
blk(QQ)     : EXISTS (QQ1:comp): ntrans(a)(QQ,QQ1)  AND Pa1 = blk(QQ1)
         OR   EXISTS (Q:stand): wtrans(throw)(QQ,Q) AND ntrans(a)(Q,Pa1),
  ..
 ENDCASES
 MEASURE Pa BY <<
```

Fig. 7. Normal transitions in PVS

introduces the measure that is used to prove the well-foundedness of the recursion, and thus termination of the function.

7 Mechanising the Semantic Relationship

Following the same steps as in the hand proofs, first a definition is given for the derived traces. A derived trace is defined as a transition of a process term by a trace, which consists of a transition by a sequence of observable events followed by a terminal transition. The terminal transitions are already defined. The transition by a sequence of observable events is defined by using the '`ntrans`' recursively over a list of normal events as follows:

```
trans_list(s:lits[normal])(P,P1:stand) : RECURSIVE bool =
CASES s OF
       null : P = P1,
cons(h,tail):EXISTS(Q:stand):ntrans(h)(P,Q) AND trans_list(tail)(Q,P1)
ENDCASES MEASURE length(s)
```

The transition of a standard process term by a trace leads it to a null (0) process, and it is defined in PVS as follows:

```
trans_trace(t:trace)(P,N:stand) : bool =
EXISTS (Q:stand):trans_list(t`1)(P,Q) AND wtrans(t`2)(Q,N) AND N = nul
```

A compensable process has both forward, and compensation behaviour. The compensation behaviour is same as that of a standard process. We mentioned earlier that it is only required to derive traces for the forward behaviour. Following the same steps as shown earlier, the derived traces for compensable processes are defined as follows:

```
 ftrans_trace(t:trace)(PP:comp,P:stand) : bool =
  EXISTS (QQ:comp): ctrans_list(t`1)(PP,QQ) AND wtrans(t`2)(QQ,P)
 ctrans_trace(tt:comp_trace)(PP:comp, N:stand) : bool =
  EXISTS (P:stand): ftrans_trace(tt`1)(PP,P)
                AND trans_trace(tt`2)(P,N) AND N = nul
```

Here, `ftrans_trace` is the derived trace for the forward behaviour, and `ctrans_list` is the compensable counterpart of `trans_list`.

By using the definitions of derived trace, we can now define the lemmas for the operators shown in Sec. 3. Lemma 3.4 and 3.5 are defined as follows:

```
s,s1,s2 : VAR list[normal]
w,w1,w2 : VAR terminal
seq_lemma : LEMMA
trans_trace((s,w))(seq(P,Q),nul) =
   EXISTS (s1,w1,s2,w2) : (s,w) = seq((s1,w1),(s2,w2)) AND
   trans_trace((s1,w1))(P,nul) AND trans_trace((s2,w2))(Q,nul)
para_lema : LEMMA trans_trace((s,w))(para(P,Q),nul) =
  EXISTS (s1,w1,s2,w2): parallel((s,w))((s1,w1),(s2,w2)) AND
  trans_trace((s1,w1))(P,nul) AND trans_trace((s2,w2))(Q,nul)
```

In the lemmas, traces are defined explicitly as pairs that make it convenient to apply induction. Induction is applied over `s` (`induct "s"`), where in the base case, `s` is empty (i.e., $t = \langle\omega\rangle$), and in the inductive case, it is a list constructed by adding an element to `s` (i.e., $t = \langle a\rangle t$). The lemmas for the other operators are defined, and proved in a similar way.

For compensable processes, lemmas are defined only for the forward behaviour. For example, Lemma 3.6 is defined in PVS as follows:

```
cpara_lemma : LEMMA
  ftrans_trace((s,w))(cpara(PP,QQ),R) = EXISTS (P,Q,s1,s2,w1,w2) :
  parallel((s,w))((s1,w1),(s2,w2)) AND ftrans_trace((s1,w1))(PP,P)
  AND ftrans_trace((s2,w2))(QQ,Q) AND R = para(P,Q)
```

The lemmas for the compensation pair, and the transaction block are crucial in our definitions, because they require mutual recursion of both standard, and compensable processes. The proof of the Lemma 3.7 involves the proof of standard processes, and the proof of the Lemma 3.8 involves the proof of compensable processes. These two lemmas are defined as follows:

```
pair_lema : LEMMA
 ctrans_trace((s,w),(s3,w3))(cpair(P,Q),nul) =
 EXISTS (s1,w1,s2,w2) : ((s,w),(s3,w3)) = pair((s1,w1),(s2,w2)) AND
    trans_trace((s1,w1))(P,nul) AND trans_trace((s2,w2))(Q,nul)
block_lema : LEMMA
trans_trace((s,w))(blk(PP),nul) =
 EXISTS (s1,w1,s2,w2):(s,w) = block((s1,w1),(s2,w2)) AND
                      ctrans_trace((s1,w1),(s2,w2))(PP,nul)
```

The inductive proofs of these two lemmas require a combined transition relation of both standard, and compensable processes, and we get the required support from the definitions of `wtrans`, and `ntrans`, where the transitions of both standard, and compensable processes are defined together. The proofs also require some additional rules that are derived from both traces, and transition rules. For brevity, we omit the definitions of lemmas of the other process terms. However, they can be defined by following the style presented in this paper. The detailed PVS proof steps and the proof trees of the lemmas can be found in [19].

8 Conclusions and Future Work

We have outlined an approach to mechanise the cCSP semantic models, and proved their relationship by using the automated theorem prover PVS. We have embedded the semantic models in PVS by using its existing logics and theories. We have defined the operators in terms of their semantic models, and laws of these operators can be proved from these semantic definitions, which helped to avoid the approach of directly encoding the laws as axioms, because it would introduce inconsistencies in the logic. The inductive proofs have followed similar steps as the steps in the hand proofs, and it has given us confidence in our language definitions. The general philosophy behind the works in [1,7,12], and ours is similar, where the work is aimed at providing an environment, where proofs about the algebra can be done in a similar level of granularity as in the hand proofs. Although defining process algebras in PVS is not new, our novel contribution is that we have not only defined the syntax, and the two semantic models, but also proved a relationship between the semantic models.

It is easy to be imprecise about recursion, and typing of the rules in the informal proofs (by hand). The mechanisation has forced us to be strict about datatypes, and recursion. This helped us to define the theorems, and the lemmas in a systematic way as well as to prove all the lemmas by following a similar fashion. The mechanisation has helped us identifying some lemmas, which were not explored in the hand proofs. The mechanisation also deepen our understanding of the semantic models for both standard and compensable processes.

We have avoided describing the semantics, and the mechanisation of synchronous composition of processes $(P \parallel_X Q)$. A separate description is needed for its definitions. We need to extend both the trace, and the operational semantics as well as the process algebra terms to define synchronisation. In synchronous composition, processes synchronise over a set of events (X), and interleave over other events. As processes synchronise over both normal, and terminal events, the existing style of separating the synchronisation of the terminal events from normal that of normal events is not applicable. Synchronising processes may fail to synchronise, and it can lead to deadlock. In such situation, instead of getting a complete behaviour, we can get only a partial behaviour from the composition. To denote it, additional terminal notation is required (e.g., $\perp$). It is also required to update the trace properties to reflect these changes. The definitions, and proofs of the semantic relationship become relatively complex, and a mechanical verification is essential to handle it. By extending the existing mechanisation, we can mechanise the semantic relationship for the synchronisation. More details can be found in [19].

We have taken a subset of the original cCSP language in our experiment. It is our future plan to extend the current experiments to include the other operators, and define the semantic relationship for them. We are also investigating to model business transactions in order to experiment the expressiveness of the language, and to improve the language features.

Acknowledgement

Thanks to the referees for suggesting improvements in the presentation of the paper.

References

[1] Archer, M. and C. L. Heitmeyer, *Human-style Theorem Proving Using PVS*, in: E. L. Gunter and A. P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97*, LNCS **1275**, 1997, pp. 33–48.

[2] Baeten, J. C. M. and W. P. Weijland, "Process Algebra," Number 18 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.

[3] Barras, B., S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi and B. Werner, *The Coq proof assistant reference manual : Version 6.1*, Technical Report 0203, INRIA (1997).

[4] Basten, T. and J. Hooman, *Process Algebra in PVS*, in: R. Cleaveland, editor, *TACAS'99*, LNCS **1579** (1999), pp. 270–284.

[5] Butler, M., T. Hoare and C. Ferreira, *A trace semactics for long-running transaction*, in: A. Abdallah, C. Jones and J. Sanders, editors, *Proceedings of 25 Years of CSP*, LNCS **3525** (2004).

[6] Butler, M. and S. Ripon, *Executable semantics for compensating CSP*, in: M. Bravetti, L. Kloul and G. Zavattaro, editors, *WS-FM 2005*, LNCS **3670** (2005), pp. 243–256.

[7] Camilleri, A. J., *Mechanizing CSP trace theory in High Order Logic*, IEEE Transactions on Software Engineering **16** (1990), pp. 993–1004.

[8] Dutertre, B. and S. Schneider, *Using a PVS embedding of CSP to verify authentication protocols*, in: E. L. Gunter and A. P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97*, LNCS **1275** (1997), pp. 121–136.

[9] Evans, N. and S. A. Schneider, *Verifying security protocols with PVS: widening the rank function approach*, Journal of Logic and Algebraic Programming **64** (2005), pp. 253–284.

[10] Gordon, M. and T. Melham, "Introduction to HOL: A Theorem Proving Environment for Higher Order Logic," Cambridge University Press, 1993.

[11] Gray, J. and A. Reuter, "Transaction Processing : Concepts and Techniques," Morgan Kaufmann Publishers, 1993.

[12] Groenboom, R., C. Hendriks, I. Polak, J. Terlouw and J. T. Udding, *Algebraic Proof Assistants in HOL*, in: *MPC '95: Mathematics of Program Construction*, LNCS **947** (1995), pp. 304–321.

[13] Hoare, C., "Communicating Sequential Process," Prentice Hall, 1985.

[14] Owre, S., J. Rushby and N. Shankar, *PVS: A Prototype Verification System*, in: D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence **607** (1992), pp. 748–752.

[15] Owre, S. and N. Shanker, *Abstract datatypes in PVS*, Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA (1993), extensively revised June 1997.

[16] Paulson, L., "Isabelle: A Generic Theorem Prover," LNCS **828**, Springer-Verlag, 1994.

[17] Plotkin, G. D., *A structural approach to operational semantics.*, Technical Report DAIMI FN-19, Aarhus University, Computer Science Department (1981).

[18] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert and J. van Tassel, *Experience with embedding hardware description languages in HOL*, in: *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience* (1993), pp. 129–156.

[19] Ripon, S., "Extending and Relating Semantic Models of Compensating CSP," Ph.D. thesis, University of Southampton (2008).

[20] Ripon, S. and M. Butler, *Relating Semantic Models of Compensating CSP*, Technical report, School of Electronics and Computer Science, University of Southampton (2006).

[21] Shankar, N. and S. Owre, *Principles and Pragmatics of Subtyping in PVS.*, in: D. Bert, C. Choppy and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT '99*, LNCS **1827** (1999), pp. 37–52.