PVS Verification of cCSP Synchronous Semantics

Shamim H. Ripon [*]

Abstract—***Compensating CSP* (cCSP) is a language defined to model long running business transactions within the framework of standard CSP process algebra. In earlier work, we have defined both traces and operational semantics of the language. We have shown the consistency between the two semantic models by defining a relationship between them. Synchronization was missing from the earlier semantic definitions which is an important feature for any process algebra. In this paper, we address this issue by extending the syntax and semantics to support synchronization and define a relationship between the semantic models. Moreover, we improve the scalability of our proof technique by mechanically verifying the semantic relationship using theorem prover PVS. We show how to embed process algebra terms and semantics into PVS and to use these embeddings to prove the semantic relationship.**

Keywords: Compensating CSP, synchronization, semantics, theorem proving, PVS

1 Introduction

Business transactions involve multiple partners coordinating and interacting with each other. These transactions have hierarchies of activities that need to be orchestrated. Business transactions also need to deal with faults that can arise at any stage of the transactions. Compensation mechanisms [1] are very important for handling faults for transactions that require a long period of time (also called *Long Running Transaction*, LRT). Process calculi are models or languages for concurrent and distributed interactive systems. Based on the framework of Hoare's CSP process algebra [2], Butler *et al* [3] introduced compensating CSP, a language to model long running transactions. The language introduces a method to declare a transaction as a process and it has constructs for orchestration of compensations.

A formal semantics offers a complete, rigorous definition of a language and provides a foundation for mathematical proofs about programs. We have defined both traces [3] and operational semantics [4] of the language. Having two semantic models of a language, it is natural to verify the consistency between them and check how they are related. We have defined a relationship between the semantic models in [5] by following a systematic approach.

Synchronization is an important and well understood feature for concurrent and distributed processes. However, synchronization was not included in our work. Based on the definitions shown in [6], in this paper we extend the cCSP semantic models to define the semantics for synchronous processes, where processes synchronize over a set of synchronizing events, and non-synchronizing processes interleave with each other. We also show that the same relationship that was defined for asynchronous processes also hold for synchronous processes. We take our work one step further by mechanical verifying the relationship by using the theorem prover PVS [7]. Mechanical verification overcomes the problem in hand proofs, also identifies potential flaws in the semantic definitions.

The rest of the paper is organized as follows. A brief overview of cCSP language is given in § 2. We then describe how the language terms are extended to define synchronization of processes in § 3. We also give an example of a web service specified by using cCSP and using the extended feature of synchronization. In the following two sections, we define how the trace and the operational semantics are extended to synchronization. § 6 defines a relationship between the semantic models and sketches the proof steps. We describe the PVS embedding of cCSP syntax and semantics in § 7. These embeddings are then used to establish the relationship between the synchronous semantic models. We outline some complimentary work in the following section. Finally, we draw our conclusions in § 9.

2 Compensating 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 categorized into *standard*, and *compensable* processes. 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. The asynchronous subset of cCSP syntax is summarized in Fig. 1.

The basic unit of the standard processes is an atomic event (A). The other operators are the sequential

[*] Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK. Tel/Fax: +44 (0)1904 434753/432767 Email: shamim@cs.york.ac.uk

(Advance online publication: 19 August 2010)

$$\begin{array}{lll|lll}
\multicolumn{3}{l}{\textbf{Standard Processes:}} & \multicolumn{3}{l}{\textbf{Compensable Processes:}} \\
P, Q ::= A & & \text{(atomic event)} & PP, QQ & ::= P \div Q & \text{(compensation pair)} \\
& \mid P \,;\, Q & \text{(sequential composition)} & & \mid PP \,;\, QQ & \\
& \mid P \,\square\, Q & \text{(choice)} & & \mid PP \,\square\, QQ & \\
& \mid P \parallel Q & \text{(parallel composition)} & & \mid PP \parallel QQ & \\
& \mid \mathit{SKIP} & \text{(normal termination)} & & \mid \mathit{SKIPP} & \\
& \mid \mathit{THROW} & \text{(throw an interrupt)} & & \mid \mathit{THROWW} & \\
& \mid \mathit{YIELD} & \text{(yield to an interrupt)} & & \mid \mathit{YIELDD} & \\
& \mid P \,\rhd\, Q & \text{(interrupt handler)} & & & \\
& \mid [PP] & \text{(transaction block)} & & &
\end{array}$$

Figure 1: cCSP syntax

$(P \;;\; Q)$, and the parallel composition $(P \parallel Q)$, the choice operator $(P \;\square\; Q)$, the interrupt handler $(P \rhd Q)$, the empty process *SKIP*, raising an interrupt *THROW*, and yielding to 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 synchronizes with yielding in another process. 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 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. 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 by pairing an empty compensation with them, e.g., $\mathit{SKIPP} = \mathit{SKIP} \div \mathit{SKIP}$.

3 Extending cCSP with Synchronization

We define a parallel operator synchronizing over observable events[1] extending our earlier definition, where processes interleave over observable events and synchronize only over terminal events[2]. We assume a set of events X over which processes will synchronize. The process $(P \parallel_X Q)$ represents the parallel composition of processes P and Q, synchronizing over the set of events X. Operationally, P and Q interact by synchronizing over the events from X, while events not in X can occur independently. An event where both processes synchronize becomes a single event in $(P \parallel_X Q)$, by a synchronizing operator which will be defined later. In the following example a business transaction is modelled by cCSP constructs added with synchronization:

Example: ***(Car Broker Web Services)*** We model a car broker web service **Broker** which provides online support to customers to negotiate car purchases and arranges loans for these. The architectural view of the web service is given in Fig. 2.

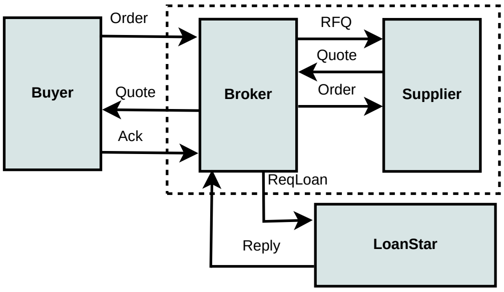

Figure 2: Architectural view of Car Broker web Services

In cCSP, a process is described in terms of its interactions with its environment or with other processes by using atomic actions. The communications are defined via channels as in standard CSP. A communication is an event described by the pair $c.v$, where c is the channel name and v is the value of the message. Input/output are defined using same construct as in CSP. Concurrent processes communicate via channels. We also use I/O parameters for compensation pair:

$$A?x \div B.x \;\;;\; P(x) \;=\; \square_{x \in S}\, A.x \div B.x \;\;;\; P(x)$$

The first step of the transaction is a compensation pair, where the primary action is to receive an order from the buyer and the compensation is to cancel the order. M is used to represent the finite set of car models ranged over by m.

$$\begin{array}{l}
\textbf{Broker} \;\widehat{=} \\
\quad (\mathit{Order}?m : M \div \mathit{CancelOrder}.m) \;;\textbf{ProcessOrder(m)}
\end{array}$$

$$\begin{array}{l}
\textbf{ProcessOrder(m)} \;\widehat{=}\;\; \mathit{RFQ}.m \;;\mathit{Quote}?q : \mathbb{F}\,Q \;; \\
\quad \square_{c \in q} \bullet \Big((\textbf{Sendorder(c)} \parallel \textbf{Loan(a)}) \parallel \textbf{SendQuote(c)} \Big)
\end{array}$$

$$\textbf{SendOrder(c)} \;\widehat{=}\; (\mathit{Order}.c \div \mathit{SKIP})$$

$$\begin{array}{ll}
\textbf{Loan(a)} \;\widehat{=} & (\mathit{ReqLoan}.a : \mathit{Amt} \div \mathit{CancelLoan}.a) \;; \\
& (\mathit{Reply}?\mathit{Accept} \;;\mathit{SKIPP} \\
& \square\; \mathit{Reply}?\mathit{Reject} \;;\mathit{THROWW})
\end{array}$$

$$\begin{array}{ll}
\textbf{SendQuote(c)} \;\widehat{=} & \mathit{Quote}.c \;;(\mathit{Ack}?\mathit{Accept} \;;\mathit{SKIPP} \\
& \qquad\qquad \square\; \mathit{Ack}?\mathit{Reject} \;;\mathit{THROWW})
\end{array}$$

The **Broker** requests the **Supplier** for available quotes (RFQ) and then selects a quote from the received quotes

[1] We use normal and observable interchangeably; normal event: $a \in \Sigma$

[2] Cause termination of a process term, a terminal event $\omega \in \Omega = \{\checkmark, !, ?\}$

(*Quote*). The **Broker** arranges a loan for the quoted car by requesting a loan from **LoanStar**. The loan amount (*Amt*) of loan to be requested is decided from the selected quote and passed to the process **Loan**. It requests loan from **LoanStar** which is either accepted or rejected. If the loan cannot be provided then an interrupt is thrown to cancel the actions that have already taken place. A compensation is added to *ReqLoan* (*CancelLoan*) so that in the case of failure in a later stage the compensation can be invoked to cancel the event. the quote is also sent to the buyer (**SendQuote**). An interrupt can be raised either by the **Buyer** by rejecting the quote or by the **LoanStar** by rejecting the requested loan. In either case, the **Supplier** will terminate yielding an interrupt thrown by the **Broker** and compensations from both **Broker** and **Supplier** will run in parallel.

The behaviour of the car broker web service is defined by combining the behaviour of **Broker, Buyer, Supplier**, and **LoanStar**, where the processes synchronize over the sets A, B and C.

$$\begin{aligned}\textbf{System} \;\widehat{=}\; & \textbf{Buyer} \parallel_A \big[\textbf{Broker} \parallel_B \textbf{Supplier}\big] \\ & \parallel_C \textbf{LoanStar}\end{aligned}$$

$$A = \{\mathit{Order}, \mathit{Quote}, \mathit{Ack}\},\; B = \{\mathit{RFQ}, \mathit{Quote}, \mathit{Order}\}$$
$$C = \{\mathit{ReqLoan}, \mathit{Reply}\}$$

The example illustrates the synchronization of processes within a transaction block, $[$**Broker** $\parallel_B$ **Supplier**$]$ and between transaction blocks (**Buyer** and **LoanStar** are transaction blocks). It also outlines how compensations are handled in each case.

4 Extended Trace Semantics

A trace records the behaviour of a process up to some moment in time. The traces of composite processes are defined in terms of their constituent processes. Processes are assumed to have an alphabet of actions Σ which does not include the terminal events $\Omega = \{\checkmark, !, ?\}$. Terminal symbols indicate the way how a process terminates. Standard processes are defined as non-empty set of traces of the form $s\langle\omega\rangle$ where $s \in \Sigma^*$ and $\omega \in \Omega$. For traces s and t, we write $s.t$ as their concatenation. Operators are first defined on traces and then lifted to set of traces to define processes. The traces of a standard process P is denoted as $T(P)$. Compensable processes consist of a set of pair of traces of the form $(p\langle\omega\rangle, p'\langle\omega'\rangle)$, where $p\langle\omega\rangle$ represents the forward behaviour and $p'\langle\omega'\rangle$ represents the compensation. $T(PP)$ denotes the trace of a compensable process PP.

Parallel processes synchronize over synchronizing events and interleave over other events. When processes fail to synchronize, the execution blocks and we get a partial behaviour from the composition. To denote partial behaviour, we assume a special terminal symbol $\perp \in \Omega$ which indicates partial trace. Partial traces are analogous to trace prefixes in standard CSP. With the definition of partial behaviour, traces from standard processes satisfy the following properties:

- $\langle\perp\rangle \in T(P)$
- $p\langle x\rangle q \in T(P) \;\Rightarrow\; p\langle\perp\rangle \in T(P) \quad (x \in \Sigma)$

We assume $\perp$ acts as a cut for trace concatenation: $p\langle\perp\rangle q = p\langle\perp\rangle$. With the introduction of the new terminal event ($\perp$), we extend the original trace definitions. The extended trace definitions for sequential operators are defined in Fig. 3.

We define a synchronization operator on events writing $A\&A'$ for the synchronization of events A and A'. Consider two processes synchronizing over events a and a', the synchronization is defined as: $a\&a = a$, and $a\&a' = \perp$ when $a \neq a'$ and do not synchronize with each other.

We define a synchronization operator over terminal events from the set Ω. Table 1 enumerates the evaluation of this operator. We also define the synchronization operator to be commutative. From Table 1 it can be seen that the operator is well-defined for all the operands in the set Ω. Case analysis shows that the synchronization operator is associative.

Table 1: Synchronization of terminal events

ω	!	!	!	?	?	✓	$\perp$
ω'	!	?	✓	?	✓	✓	ω
$\omega\&\omega'$	!	!	!	?	?	✓	$\perp$

Assuming $a, a' \in X$ and $b, b' \notin X$, the parallel composition of traces from standard processes are defined as follows:

$$\begin{array}{lcl}
\langle\omega\rangle \parallel_X \langle\omega'\rangle & = & \{\, \langle\omega\&\omega'\rangle \,\} \\
\langle a\rangle p \parallel_X \langle\omega\rangle & = & \{\, \langle\perp\rangle \,\} \\
\langle a\rangle p \parallel_X \langle a'\rangle q & = & \{\, (a\&a')r \mid r \in (p \parallel_X q) \,\} \\
\langle b\rangle p \parallel_X \langle\omega\rangle & = & \{\, \langle b\rangle r \mid r \in (p \parallel_X \langle\omega\rangle) \,\} \\
\langle b\rangle p \parallel_X \langle a\rangle q & = & \{\, \langle b\rangle r \mid r \in (p \parallel_X \langle a\rangle q) \,\} \\
\langle b\rangle p \parallel_X \langle b'\rangle q & = & \{\, \langle b\rangle r \mid r \in (p \parallel_X \langle b'\rangle q) \,\} \\
 & & \cup \{\, \langle b'\rangle r \mid r \in (\langle b\rangle p \parallel_X q) \,\}
\end{array}$$

The parallel and synchronization operators are symmetric. For brevity we omit the symmetric cases. The parallel composition of standard processes is defined as follows:

$$\begin{aligned} T(P \parallel_X Q) \;=\; & \{\, r \mid r \in (p \parallel_X q) \\ & \wedge\; p \in T(P) \;\wedge\; q \in T(Q) \,\} \end{aligned}$$

With the definition of partial behaviour ($\perp$), a pair of traces $(p\langle\omega\rangle, p'\langle\omega'\rangle)$ of a compensable process satisfies the following properties: For $x \in \Sigma$,

Atomic Action:

For $A \in \Sigma$ $T(A) = \{\langle\bot\rangle, \langle A, \checkmark\rangle, \langle A, \bot\rangle\}$

Basic Processes:

$T(SKIP) = \{\langle\checkmark\rangle, \langle\bot\rangle\}$, $T(THROW) = \{\langle!\rangle, \langle\bot\rangle\}$,
$T(YIELD) = \{\langle?\rangle, \langle\checkmark\rangle, \langle\bot\rangle\}$

Choice: $T(P \Box Q) = T(P) \cup T(Q)$

Sequential Composition:

$p\langle\checkmark\rangle\ ;\ q = p.q,\ \ p\langle\omega\rangle\ ;\ q = p\langle\omega\rangle$, where $\omega \neq \checkmark$
$T(P\ ;\ Q) = \{p\ ;\ q \mid p \in T(P) \wedge q \in T(Q)\}$

Interrupt Handler:

$p\langle!\rangle \rhd q = p.q,\ \ p\langle\omega\rangle \rhd q = p\langle\omega\rangle$ where $\omega \neq !$
$T(P \rhd Q) = \{p \rhd q \mid p \in (P) \wedge q \in T(Q)\}$

(a) Standard

Choice: $T(PP \Box PQ) = T(PP) \cup T(QQ)$

Sequential Composition:

$(p\langle\checkmark\rangle, p')\ ;\ (q, q') = (pq, q'\ ;\ p')$
$(p\langle\omega\rangle, p')\ ;\ (q, q') = (p\langle\omega\rangle, p')$ where $\omega \neq \checkmark$
$T(PP\ ;\ QQ) = \{pp\ ;\ qq \mid pp \in T(PP) \wedge qq \in T(QQ)\}$

Compensation Pair:

$p\langle\checkmark\rangle \div q = (p\langle\checkmark\rangle, q)$ and
$p\langle\omega\rangle \div q = (p\langle\omega\rangle, \langle\checkmark\rangle), (p\langle\omega\rangle, \langle\bot\rangle)$ where $\omega \neq \checkmark$
$T(P \div Q) = \{(\langle?\rangle, \langle\checkmark\rangle)\} \cup \{p \div q \mid p \in T(P) \wedge q \in T(Q)\}$

Transaction Block:

$[p\langle!\rangle, p'] = p.p',\ [p\langle\checkmark\rangle, p'] = p\langle\checkmark\rangle,\ [p\langle\bot\rangle, p'] = p\langle\bot\rangle$
$T([PP]) = \{[p, p'] \mid (p, p') \in T(PP)\}$

(b) Compensable

Figure 3: Trace semantics of sequential processes

– $(\langle\bot\rangle, p') \in T(PP)$

– $(p\langle x\rangle q, p') \in T(PP) \Rightarrow (p\langle\bot\rangle, \_) \in T(PP)$

– $(p, p'\langle x\rangle q') \in T(PP) \Rightarrow (p, p'\langle\bot\rangle) \in T(PP)$

The trace semantics for compensable parallel processes is defined as follows:
$(p, p') \parallel_X (q, q') =$
$\{(r, r') \mid r \in (p \parallel_X q) \wedge r' \in (p' \parallel_X q') \wedge last(r) \neq \bot\}$
$\cup \{(r, \langle\bot\rangle) \mid r \in (p \parallel_x q) \wedge last(r) = \bot\}$

$$\begin{aligned} T(PP \parallel_X QQ) \ &= \ \{\, rr \mid rr \in (pp \parallel_X qq) \\ &\quad \wedge pp \in T(PP) \wedge qq \in T(QQ)\,\} \end{aligned}$$

$last(t)$ returns the terminal symbol from a trace t.

5 Extended Operational Semantics

The operational semantics are defined by using labelled transition systems [8]. Inference rules are used to define the transitions that a process may perform, which for composite processes are given in terms of the possible transition of the constituents (See [4] for detail). Two types of transition rules are defined: normal and terminal. Normal transition is caused by a normal event resulting in a transition of a process term from one state to another. Terminal transition is caused by a terminal event where standard process terms terminate to a null process and the forward behaviour of compensable process terms terminate leaving the attached compensation for future reference. Note that the language terms are extended to define the null (0) process that cannot perform any action. For standard and compensable process terms P and PP (where $P, PP \neq 0$), the normal and terminal transitions are defined as followed:

$$\begin{array}{rl} P \xrightarrow{a} P', & PP \xrightarrow{a} PP' \quad (a \in \Sigma) \\ P \xrightarrow{\omega} 0, & PP \xrightarrow{\omega} P \ \ (\omega \in \{\checkmark, !, ?\}) \\ & (P \text{ is the compensation of } PP) \end{array}$$

We extend the transition rules by defining the transitions by a $\bot$ where both standard and compensable processes terminate to a null process. For any process terms P and PP (where $P, PP \neq 0$), the transitions by a $\bot$ are defined as follows:

$$P \xrightarrow{\bot} 0, \qquad PP \xrightarrow{\bot} 0 \tag{1}$$

The transition rules defined in equation (1) cover the transitions for both standard and compensable process terms by the $\bot$. Hence we do not need to define additional transition rules by a $\bot$. The transition rules for sequential standard and compensable processes are defined in Fig. 4(a) and Fig. 4(b) respectively.

As $\bot$ is introduced during process synchronization and $\bot$ is a useful semantic device that helps us deriving semantic correspondence, we define the extended transition rules for parallel processes and define those transitions that introduce a $\bot$. For a compensable process the transition by a $\bot$ lead to a null process and according to our definition no compensations are stored (being partial behaviour). The transition rules for standard and compensable parallel processes are shown in Fig. 5(a) and Fig. 5(b) respectively.

6 Semantic Relationship

Over the years, several techniques have been used to establish relationship between different semantic models. Widely used techniques are deriving one semantics from another (e.g. [9, 10]), extracting the behaviour from one semantic model and showing its relation with another

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

Basic Processes:

$\mathit{SKIP} \xrightarrow{\checkmark} 0,\ \mathit{THROW} \xrightarrow{!} 0,\ \mathit{YILED} \xrightarrow{?} 0,\ \mathit{YIELD} \xrightarrow{\checkmark} 0$

Sequential Composition:

$$\frac{P \xrightarrow{a} P'}{(P\ ;\ Q) \xrightarrow{a} (P'\ ;\ Q)} \quad \frac{P \xrightarrow{\checkmark} 0 \wedge Q \xrightarrow{\alpha} Q'}{(P\ ;\ Q) \xrightarrow{\alpha} Q'} \quad \frac{P \xrightarrow{\omega} 0}{(P\ ;\ Q) \xrightarrow{\omega} 0}\ (\omega \neq \checkmark)$$

Choice:

$$\frac{P \xrightarrow{\alpha} P'}{P \,\Box\, Q \xrightarrow{\alpha} P'} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \,\Box\, Q \xrightarrow{\alpha} Q'}\ (\alpha \in \Sigma \cup \Omega)$$

Interrupt handler:

$$\frac{P \xrightarrow{a} P'}{P \,\rhd\, Q \xrightarrow{a} P' \,\rhd\, Q} \quad \frac{P \xrightarrow{!} 0 \,\wedge\, Q \xrightarrow{\alpha} Q'}{P \,\rhd\, Q \xrightarrow{\alpha} Q'} \quad \frac{P \xrightarrow{\omega} 0}{P \,\rhd\, Q \xrightarrow{\omega} 0}\ (\omega \neq\, !)$$

(a) Standard

Choice:

$$\frac{PP \xrightarrow{a} PP'}{PP \,\Box\, QQ \xrightarrow{a} PP'} \quad \frac{QQ \xrightarrow{a} QQ'}{PP \,\Box\, QQ \xrightarrow{a} QQ'} \quad \frac{PP \xrightarrow{\omega} P}{PP \,\Box\, QQ \xrightarrow{\omega} P} \quad \frac{QQ \xrightarrow{\omega} Q}{PP \,\Box\, QQ \xrightarrow{\omega} Q}$$

Sequential Composition:

$$\frac{PP \xrightarrow{a} PP'}{PP\ ;\ QQ \xrightarrow{a} PP'\ ;\ QQ} \quad \frac{PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{\omega} Q}{PP\ ;\ QQ \xrightarrow{\omega} Q\ ;\ P} \quad \frac{PP \xrightarrow{\omega} P}{PP\ ;\ QQ \xrightarrow{\omega} P}\ (\omega \neq \checkmark)$$

$$\frac{PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{a} QQ'}{PP\ ;\ QQ \xrightarrow{a} \langle QQ', P\rangle} \quad \frac{QQ \xrightarrow{a} QQ'}{\langle QQ, P\rangle \xrightarrow{a} \langle QQ', P\rangle} \quad \frac{QQ \xrightarrow{\omega} Q}{\langle QQ, P\rangle \xrightarrow{\omega} Q\ ;\ P}$$

Compensation Pair:

$$\frac{P \xrightarrow{a} P'}{P \div Q \xrightarrow{a} P' \div Q} \quad \frac{P \xrightarrow{\checkmark} 0}{P \div Q \xrightarrow{\checkmark} Q} \quad \frac{P \xrightarrow{\omega} 0}{P \div Q \xrightarrow{\omega} \mathit{SKIP}} \quad (\omega \neq \checkmark)$$

Transaction Block:

$$\frac{PP \xrightarrow{a} PP'}{[PP] \xrightarrow{a} [PP']} \quad \frac{PP \xrightarrow{\checkmark} P}{[PP] \xrightarrow{\checkmark} 0} \quad \frac{PP \xrightarrow{!} P \,\wedge\, P \xrightarrow{\alpha} P'}{[PP] \xrightarrow{\alpha} P'}\ (\alpha \in \Sigma \cup \Omega)$$

(b) Compensable

Figure 4: Operational Semantics for sequential processes

$$\frac{P \xrightarrow{\omega} 0 \wedge Q \xrightarrow{\omega'} 0}{P \,\|_X\, Q \xrightarrow{\omega \& \omega'} 0} \quad \frac{p \xrightarrow{a} P' \,\wedge\, Q \xrightarrow{a'} Q'}{P \,\|_X\, Q \xrightarrow{a \& a'} P' \,\|_X\, Q'}\ (a, a' \in X)$$

$$\frac{P \xrightarrow{a} P' \,\wedge\, Q \xrightarrow{\omega} 0}{P \,\|_X\, Q \xrightarrow{\perp} 0} \quad \frac{P \xrightarrow{\omega} 0 \,\wedge\, Q \xrightarrow{a} Q'}{P \,\|_X\, Q \xrightarrow{\perp} 0}\ (a \in X)$$

$$\frac{P \xrightarrow{b} P'}{P \,\|_X\, Q \xrightarrow{b} P' \,\|_X\, Q} \quad \frac{Q \xrightarrow{b} Q'}{P \,\|_X\, Q \xrightarrow{b} P \,\|_X\, Q'}\ (b \notin X)$$

(a) Standard

$$\frac{PP \xrightarrow{b} PP'}{PP \,\|_X\, QQ \xrightarrow{b} PP' \,\|_X\, QQ} \quad \frac{PP \xrightarrow{a} PP' \,\wedge\, QQ \xrightarrow{a'} QQ'}{PP \,\|_X\, QQ \xrightarrow{a \& a'} PP' \,\|_X\, QQ'}\ (a \,\&\, a' \neq \perp)$$

$$\frac{PP \xrightarrow{\omega} P \,\wedge\, QQ \xrightarrow{\omega'} Q}{PP \,\|_X\, QQ \xrightarrow{\omega \& \omega'} P \,\|_X\, Q}\ (\omega \,\&\, \omega' \neq \perp) \quad \frac{PP \xrightarrow{\omega} P \,\wedge\, QQ \xrightarrow{\omega'} Q}{PP \,\|_X\, QQ \xrightarrow{\omega \& \omega'} 0}\ (\omega \,\&\, \omega' = \perp)$$

$$\frac{PP \xrightarrow{a} PP' \,\wedge\, QQ \xrightarrow{\omega} Q}{PP \,\|_X\, QQ \xrightarrow{\perp} 0} \quad \frac{PP \xrightarrow{a} PP' \,\wedge\, QQ \xrightarrow{a'} QQ'}{PP \,\|_X\, QQ \xrightarrow{a \& a'} 0}\ (a \,\&\, a' = \perp)$$

(b) Compensable

Figure 5: Operational Semantics for synchronous processes

(e.g. [11]) etc. Roscoe [12] outlines how to define the semantic relationship for CSP. In our earlier work [5, 13], we have adopted a systematic approach showing a relationship between the semantic models. Traces are extracted from the transition rules of the operational semantics and show that the extracted traces correspond to the original traces for each term of the language and finally, prove the correspondence by structural induction over the process terms. The steps are depicted in Fig. 6.

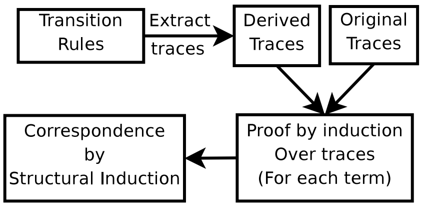
Transition Rules
Extract traces
Derived Traces
Original Traces
Proof by induction Over traces (For each term)
Correspondence by Structural Induction

Figure 6: Steps for semantic correspondence

In this paper, we extend our earlier approach to define and prove the relationship between the synchronous semantic models. Due to the introduction of partial behaviour, proving the correspondence for synchronous semantic modes becomes critical. We briefly describe the steps shown in Fig. 6 for asynchronous processes and extend those steps for synchronous processes.

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

$$\begin{aligned} P \xrightarrow{\langle \omega \rangle} Q &= P \xrightarrow{\omega} Q \\ P \xrightarrow{\langle a \rangle t} Q &= \exists\, P' \cdot P \xrightarrow{a} P' \,\wedge\, P' \xrightarrow{t} Q \end{aligned}$$

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

Definition 1. For a trace t, $t \in DT(P) \,=\, P \xrightarrow{t} 0$

For compensable processes, it is required to extract traces from both forward and compensation behaviour. First, we define the lifted forward behaviour and then add the behaviour of compensation by reusing the above definition. For a compensable process PP, we get the following definition:

Definition 2. For traces t and t',

$$\begin{aligned} (t, t') \in DT(PP) &= PP \xrightarrow{(t,t')} 0 \\ &= \exists\, P' \cdot PP \xrightarrow{t} P' \wedge P \xrightarrow{t} 0 \end{aligned}$$

Finally, the semantic relationship is defined as follows:

Theorem 1. *For a standard process term P ($P \neq 0$),*

$$DT(P) \,=\, T(P)$$

For a compensable process terms PP, where $PP \neq 0$

$$DT(PP) \,=\, T(PP)$$

The theorem is proved by showing that

$$\begin{aligned} t \in DT(P) &= t \in T(P) \\ (t, t') \in DT(PP) &= (t, t') \in T(PP) \end{aligned}$$

We apply induction over process terms and define supporting lemmas for 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,

$$t \in DT(P \otimes Q) \quad = \quad t \in T(P \otimes Q) \qquad (2)$$

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

$$(t, t') \in DT(PP \otimes QQ) = (t, t') \in T(PP \otimes QQ) \qquad (3)$$

Consider the sequential composition of processes P and Q. By using (2), the semantic relationship is shown by,

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

From Def. 1, we get the following equation,

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

We also expand the definition of trace semantics as follows:

$$\begin{aligned} & t \in \ T(P \ ; Q) \\ & \quad = \exists\, p, q \cdot t = (p \ ; \ q) \ \wedge \ p \in T(P) \ \wedge \ q \in T(Q) \\ & \quad = \exists\, p, q \cdot t = (p \ ; \ q) \ \wedge \ p \in DT(P) \ \wedge \ q \in DT(Q) \\ & \quad = \exists\, p, q \cdot t = (p \ ; \ q) \ \wedge \ P \xrightarrow{p} 0 \ \wedge \ Q \xrightarrow{q} 0 \end{aligned}$$

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

Lemma 1.
$(P \ ; \ Q) \xrightarrow{t} 0 = \exists\, p, q \cdot t = (p \ ; \ q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

The lemma is proved by applying induction over the trace t, where $t = \langle \omega \rangle$ is the base case, and $t = \langle a \rangle t$ is the inductive case. Similarly, the supporting lemmas for all the other terms of the language are defined and proved.

For synchronous processes, we follow the same approach added with the newly defined $\perp$ event. With the introduction of partial behaviour, the definition of derived traces remains the same except for the compensable processes. For a pair of traces (t and t'), the derived traces of synchrnous compensable processes is defined as follows:

$$PP \xrightarrow{(t,t')} 0 \ = \ \begin{cases} \exists\, R \cdot PP \xrightarrow{t} R \wedge R \xrightarrow{t'} 0 & last(t) \neq \perp \\ PP \xrightarrow{t} 0 \wedge t' = \langle \perp \rangle & last(t) = \perp \end{cases}$$

Considering Theorem 1, for synchronous processes we prove the following lemma:

Lemma 2. *For standard process terms P and Q,*

$$DT(P \parallel_X Q) \quad = \quad T(P \parallel_X Q)$$

For compensable process terms PP and QQ,

$$DT(PP \parallel_X QQ) \quad = \quad T(PP \parallel_X QQ)$$

By following the approach shown earlier we formulate the following lemma for standard processes:

Lemma 3. $(P \parallel_X \ Q) \xrightarrow{t} 0 = \ \exists\, p, q \cdot t \in (p \parallel_X q) \wedge P \xrightarrow{p} 0 \wedge Q \xrightarrow{q} 0$

Based on the scenario when synchronizing processes fail to synchronize and return partial behaviour, we state two separate lemmas. First, we assume that there is no failure during the synchronization of processes:

Lemma 4. $(PP \parallel_X QQ) \xrightarrow{t} R =$
$\exists\, p, q, P, Q \cdot t \in (p \parallel_X q) \wedge last(t) \neq \perp$
$\wedge\, PP \xrightarrow{p} P \wedge QQ \xrightarrow{q} Q \wedge R = (P \parallel_X Q)$

The following lemma is defined for the cases when the synchronizing processes fail to synchronize:

Lemma 5. $(PP \parallel_X QQ) \xrightarrow{t} 0 =$
$\exists\, p, q \cdot t \in (p \parallel_X q) \wedge last(t) = \perp$
$\wedge\, p \in T(PP) \wedge q \in T(QQ)$

In earlier work [14], we have shown how to mechanically proof the relationship between the asynchronous semantic models by embedding the cCSP syntax and semantic models into the theorem prover PVS, where the mechanical proofs have followed the similar proof steps as in hand proofs shown in [5]. After extending the semantic models to synchronization, instead of proving the relationship by hand, we directly prove them by using PVS. In the following section, we describe how we define and prove the semantic relationship for synchronous models by extending the asynchronous embeddings in PVS.

7 Mechanizing Relationship

An embedding is a semantic encoding of one specification language into another, especially, to reuse the existing tools of the target language. Mechanization steps of synchronous processes are outlined in this paper. Detail mechanization steps are described in [13]. PVS mechanization steps are sketched in Fig. 7.

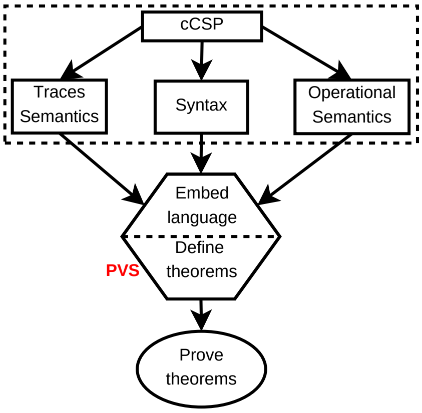

Figure 7: PVS mechanization steps

7.1 cCSP Syntax

First, we define the cCSP syntax. Separate notation is used to define the standard and compensable processes. As PVS supports overloading, same notations can be used for the operational and the trace semantics. Fig. 8 summarizes the PVS definition of asynchronous subset 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)`			

Figure 8: cCSP syntax in PVS

The syntax is then extended to define the terms for synchronization. To denote the trace semantics, we write `full_parallel(X)(P,Q)` ($P \parallel_X Q$) for standard processes and `cfull_parallel(X)(PP,QQ)` ($PP \parallel_X QQ$) for compensable processes.

7.2 Process Algebra Terms

Proofs about properties of a process algebra often use induction on the structure of the algebra. PVS has a mechanism called abstract datatype [15], for which PVS generated an induction scheme, and it is convenient to model process algebra terms as an abstract datatype. cCSP has standard, and compensable process terms and importantly, these process terms are mutually dependant on each other. Mutually recursive datatype is not directly admissible by PVS. However, PVS has an extended support of *sub-datatype* [15, 16], 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. By using this facility we define cCSP process algebra terms as follows:

```
pa_terms   : DATATYPE WITH SUBTYPES stand, comp
 BEGIN
  Skip     : skip?    : stand
  choice(P: stand, Q: stand)      : choice?    : stand
  seq(P:stand, Q:stand)           : seq?       : stand
  |>(P: stand, Q: stand)          : inthnd?    : stand
  cseq(PP : comp, QQ : comp)      : c_seq?     : comp
  cchoice(PP : comp, QQ : comp)   : c_choice?  : comp
  cpair(P: stand, Q : stand)      : cpair?     : comp
  blk(PP : comp)                  : blk?       : stand
  synpara(X:setof[normal],P:stand,Q:stand)
                                  :synpara?    : stand
  csynpara(X:setof[normal],PP:comp, QQ:comp)
                                  :csynpara?   : comp
  ...% other terms are omitted from this presentation
END pa_terms
```

`synpara` and `csynpara` are the extended definitions for the synchronous process terms. 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 processes of types '`stand`' and '`comp`'.

7.3 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. For synchronous processes, we first define the synchronization of terminal evens shown in Table 1 by extending the asynchronous definition (`parallel`).

```
syn_parallel(w3:terminal)(w1,w2:terminal):bool=
 IF w3 = bottom THEN
    w1 = bottom OR w2 = bottom
 ELSE parallel(w3)(w1,w2) ENDIF
```

The trace semantics for synchronous processes are then defined by following the definitions shown in Sec. 4. First we define operators over traces then lift it over set of traces to define processes. The trace semantics of both standard and compensable processes are defined in PVS as follows:

```
full_parallel(X)((s1,w1))((s2,w2))((s3,w3)):RECURSIVE bool=
 CASES s3 OF
 null:null?(s1) AND null?(s2)  AND syn_parallel(w3)(w1,w2)
  OR cons?(s1) AND X(car(s1)) AND null?(s2) AND w3 = bottom
  OR cons?(s2) AND X(car(s2)) AND null?(s1) AND w3 = bottom
  OR cons?(s1) AND X(car(s1)) AND cons?(s2) AND X(car(s2))
               AND  car(s1) /= car(s2) AND w3 = bottom,
 cons(a,tail):
  IF X(a) THEN cons?(s1) AND cons?(s2)     AND
             car(s1) = a AND car(s2) = a AND
    full_parallel(X)((cdr(s1),w1))((cdr(s2),w2))((tail,w3))
  ELSE cons?(s1) AND car(s1) = a AND
```

```
        full_parallel(X)((cdr(s1),w1))((s2,w2))((tail,w3))
     OR cons?(s2) AND car(s2) = a AND
        full_parallel(X)((s1,w1))((cdr(s2),w2))((tail,w3))
  ENDIF ENDCASES
 MEASURE length(s3)
full_parallel(X)(P,Q : process): process =
{t : trace | EXISTS (p:(P),q:(Q),s1,w1,s2,w2,s3,w3):
    p = (s1,w1) AND q = (s2,w2) AND t = (s3,w3) AND
    full_parallel(X)((s1,w1))((s2,w2))((s3,w3)) }

cfull_parallel(X)((p,p1))((q,q1))((r,r1)) : bool =
    (full_parallel(X)(p)(q)(r) AND
    full_parallel(X)(p1)(q1)(r1) AND r‘2 /= bottom)
 OR full_parallel(X)(p)(q)(r) AND
    r‘2 = bottom AND null?(r1‘1) AND r1‘2 = bottom
 cfull_parallel(X)(PP,QQ:comp_process):comp_process=
 { tt:comp_trace | EXISTS (pp:(PP),qq:(QQ)) :
                  cfull_parallel(X)(pp)(qq)(tt) }
```

We represent traces as a pair: `(s,w)`, where `s` is the sequence of normal events and `w` is the terminal event.

7.4 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 equations derived from the transition rules. 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. The terminal transition for the process terms are defines as `wtrans` and the normal transitions are defined as `ntrans` (See [13],[14] for details). We then define the transition rules for synchronous processes by following the definitions given in Fig. 5(a) and 5(b).

In a normal transition, processes either synchronize or interleave. By extending the transition rules of asynchronous processes we defne the transition rules for synchronous processes as follows:

```
synpara(X,Q,R):
 IF X(a) THEN
  EXISTS Q1,R1 : ntrans(a)(Q,Q1) AND  ntrans(a)(R,R1) AND
              Pa1 = synpara(X,Q1,R1)
 ELSE EXISTS Q1: ntrans(a)(Q,Q1) AND Pa1 = synpara(X,Q1,R)
   OR EXISTS R1: ntrans(a)(R,R1) AND Pa1 = synpara(X,Q,R1)
 ENDIF
csynpara(X,QQ,RR) :
IF X(a) THEN
 EXISTS QQ1,RR1:ntrans(a)(QQ,QQ1) AND ntrans(a)(RR,RR1) AND
                 Pa1 = csynpara(X,QQ1,RR1)
 ELSE
 EXISTS QQ1:ntrans(a)(QQ,QQ1) AND Pa1 = csynpara(X,QQ1,RR)
 OR EXISTS RR1:ntrans(a)(RR,RR1) AND Pa1= csynpara(X,QQ,RR1)
```

The terminal transitions are defined as follows:

```
synpara(X,Q,R):
    EXISTS w1,w2: syn_wtrans(w1)(Q,nul) AND
                  syn_wtrans(w2)(R,nul) AND
                  syn_parallel(w)(w1,w2) AND P1 = nul
 OR EXISTS (a:normal,w1,Q1): X(a) AND ntrans(a)(Q,Q1) AND
    syn_wtrans(w1)(R,nul) AND w = bottom AND P1 = nul
 OR EXISTS (a:normal,w1,R1) : X(a) AND ntrans(a)(R,R1) AND
    syn_wtrans(w1)(Q,nul) AND w = bottom AND P1 = nul
 OR EXISTS (a1,a2:normal,Q1,R1):
               X(a1) AND X(a2) AND a1 /= a2 AND
               ntrans(a1)(Q,Q1) AND ntrans(a2)(R,R1) AND
                w = bottom AND P1 = nul,
csynpara(X,QQ,RR):
    EXISTS Q1,R1,w1,w2 : syn_wtrans(w1)(QQ,Q1) AND
        syn_wtrans(w2)(RR,R1) AND syn_parallel(w)(w1,w2)
        AND w /= bottom AND P1 = synpara(X,Q1,R1)
 OR EXISTS (a:normal,w1,QQ1,R1): X(a) AND
         ntrans(a)(QQ,QQ1) AND syn_wtrans(w1)(RR,R1) AND
         w = bottom AND P1= nul
 OR EXISTS (a:normal,w1,Q1,RR1): X(a) AND
         syn_wtrans(w1)(QQ,Q1) AND ntrans(a)(RR,RR1) AND
         w = bottom and P1 = nul
 OR EXISTS (a1,a2:normal,QQ1,RR1):
        X(a1) AND X(a2) AND a1 /= a2 AND
        ntrans(a1)(QQ,QQ1) AND ntrans(a2)(RR,RR1) AND
        w = bottom AND P1 = nul
```

7.5 Semantic Relationship

By following Def. 1, the derived traces for standard processes are defined as '`trans_trace`'. It defines the transition of a process by a trace consisting of a transition by a sequence of normal events followed by transition by a terminal event. Consider a trace t, where $t = t'\langle\omega\rangle$.

$$P \xrightarrow{t'\langle\omega\rangle} 0 \quad = \quad \exists\, P' \cdot P \xrightarrow{t'} P' \wedge P' \xrightarrow{\omega} 0$$

We then define Lemma 3 by using the definition of both derived traces and trace rules as follows:

```
synpara_lemma : LEMMA
  trans_trace((s,w))(synpara(X,P,Q),nul) =
   EXISTS (s1,w1,s2,w2) :
    full_parallel(X)((s1,w1))((s2,w2))((s,w)) AND
    trans_trace((s1,w1))(P,nul)               AND
    trans_trace((s2,w2))(Q,nul)
```

For compensable processes, we only need to prove that the lifted forward behaviour corresponds to the original traces and reuse the proofs of standard processes for compensations. The definition of derived traces shown in Def. 2 consists of the derived trace of both forward and compensation behaviour. To prove our lemmas (Lemma 4 and 5) we only need to define the forward behaviour and it is defined as `ftrans_trace` ($PP \xrightarrow{t} P$).

First, we define the lemma considering the processes will not fail to synchronize and hence, there is no bottom event in the derived traces:

```
csynpara_lemma : LEMMA
 ftrans_trace((s,w))(csynpara(X,PP,QQ),R) =
   EXISTS (s1,w1,s2,w2,P,Q): w /= bottom AND
   full_parallel(X)((s1,w1))((s2,w2))((s,w)) AND
   ftrans_trace((s1,w1))(PP,P) AND
   ftrans_trace((s2,w2))(QQ,Q) AND
   R = synpara(X,P,Q)
```

Next, we define the lemma where compensable processes fail to synchronize during their synchronization. The main difference is that the derived trace now ends with a $\perp$ representing the partial behaviour, and compensations are not accumulated after termination.

```
lema_bot : LEMMA
 ftrans_trace((s,w))(csynpara(X,PP,QQ),nul) =
  EXISTS (s1,w1,s2,w2,P,Q):  w = bottom AND
   full_parallel(X)((s1,w1))((s2,w2))((s,w)) AND
   ftrans_trace((s1,w1))(PP,P)               AND
   ftrans_trace((s2,w2))(QQ,Q)
```

All these lemmas are proved interactively by applying induction over traces `((s,w))`. PVS has a strong support for induction scheme which facilities proving such lemmas.

8 Related Work

One of the contributions most related to our work is by Basten and Hooman in [17], where the focus is on the use of a general purpose proof checker, e.g., tool support for the proof of theoretical properties of an ACP-style process algebra [18] . The idea is to apply equational reasoning. Mechanical support for both verification of concrete applications and proving theoretical properties of the process algebra are investigated.

PVS has been used in [19, 20] to mechanize the trace semantics of CSP. Their goal is to verify an authentication protocol specified in CSP to overcome errors in the manual verification as well as improve the scalability of the approach. The mechanization is based on a semantic embedding of CSP. The traces are defined by using a list of events and processes are defined by prefix-closed sets of traces. The important distinction with the present work is that cCSP traces are non-empty and completed and processes are defined accordingly.

Camilleri [21] showed how to mechanize a subset of the CSP operators by using the theorem prover HOL [22]. The trace model for a subset of the CSP operators was mechanized in HOL. Initially, events, alphabets and traces are defined and then CSP operators are defined in terms of their trace semantic models. And later laws related to the operators are proved from the sematic definition. In contrast to our approach no syntax is defined at this stage and operators are defined directly in HOL. Syntax is defined later and the semantics of the language is shown based on the already defined semantics. A similar work for the π-calculus can be found in [23]. One of our main goals is to explore the ways of incorporating process algebra in a general purpose theorem prover. In that respect, a closely related research on the tool support for a process algebra shown in [24], where a CSP-like algebra, called DI-Algebra [25] is formalized in HOL. The algebra is used to reason about synchronous circuits. Process syntax and algebraic laws are defined, but no semantics are defined.

9 Concluding Remarks

We have extended cCSP language to define synchronization. We introduced the notion of partial behaviour which allows to model the behaviour of synchronous processes that fail to synchronize. The formal foundation of the language is strengthen by establishing a relationship between the semantic models by showing that traces extracted from the operational semantics correspond to the original trace semantics. Demonstrating the relationship between these two semantics of the ensures the consistency of the semantic description of the language.

We have started mechanizing the semantic models and their relationship in order to investigate the feasibility of the mechanization process. We have achieved our goal by successfully proving the semantic relationship for the synchronous processes. Defining process algebras in PVS is not new a new idea. The novelty of this experiment is that, we have not only defined the cCSP process algebra, and the two semantic models, but we have also mechanically proved a relationship between these semantic models.

In the hand proofs, it is easy to be imprecise about recursion, and typing of the rules. The mechanization forces to be strict about datatypes, and recursion. This helped us to define the theorems, and the lemmas in a systematic way, and to prove all the lemmas by following a similar fashion. The mechanization also helped us identifying some lemmas which were not explored earlier. The mechanization of the semantic models and their relationships also deepen our understanding of the semantic models for both standard and compensable processes.

Having a firm grasp of the semantic models, we are now in a better position to extend the language by defining some important operators for the process algebra, such as event hiding, recursion, distinction between external and internal choice in combination with compensations. In standard CSP, the distinction between the two choice operators is achieved by using the Failure/Divergences model which can serve as the basis for our work on cCSP. Our future plan also includes developing a tool support for cCSP which will allow model check as well as animate the specifications.

Acknowledgement

The research has been carried out at the University of Southampton, UK as a part the author's D.Phil project. Thanks to my supervisor Michael Butler for his constant support throughout the project.

References

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

[2] C. Hoare, *Communicating Sequential Process.* Prentice Hall, 1985.

[3] M. Butler, T. Hoare, and C. Ferreira, “A trace semactics for long-running transaction,” in *Proceedings of 25 Years of CSP*, ser. LNCS, A. Abdallah, C. Jones, and J. Sanders, Eds., vol. 3525. London: Springer-Verlag, 2004.

[4] M. Butler and S. Ripon, “Executable semantics for compensating CSP,” in *WS-FM 2005*, ser. LNCS, M. Bravetti, L. Kloul, and G. Zavattaro, Eds., vol. 3670. Springer-Verlag, September 1-3 2005, pp. 243–256.

[5] S. Ripon and M. Butler, “Relating Semantic Models of Compensating CSP,” School of Electronics and Computer Science, University of Southampton, Tech. Rep., May 2006.

[6] M. Butler and T. Hoare, “Towards refinement of joint transaction,” [Unpublished draft].

[7] S. Owre, J. Rushby, and N. Shankar, “PVS: A Prototype Verification System,” in *11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Springer-Verlag, June 1992, pp. 748–752.

[8] G. D. Plotkin, “A structural approach to operational semantics.” Aarhus University, Computer Science Department, Tech. Rep. DAIMI FN-19, September 1981.

[9] C. Hoare and H. Jifeng, *Unifying Theories of Programming.* Prentice Hall International Series in Computer Science, 1998.

[10] H. Zhu, J. P. Bowen, and J. He, “From operational semantics to denotational semantics for Verilog,” in *CHARME 2001*, ser. LNCS, T. Margaria and T. F. Melham, Eds., vol. 2144, 2001, pp. 449–466.

[11] S. Schneider, “An operational semantics for timed CSP,” *Journal of Information and computing*, vol. 116, no. 2, pp. 193–213, 1995.

[12] A. Roscoe, *The Theory and Practice of Concurrency*, pearson ed. Prentice Hall, 1998.

[13] S. Ripon, “Extending and Relating Semantic Models of Compensating CSP,” Ph.D. dissertation, University of Southampton, 2008.

[14] S. H. Ripon and M. J. Butler, “PVS Embedding of cCSP Semantic Models and their Relationship,” *Electronic Notes in Theoretical Computer Science*, vol. 250, pp. 103–118, 2009.

[15] S. Owre and N. Shanker, “Abstract datatypes in PVS,” Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep. SRI-CSL-93-9R, December 1993, extensively revised June 1997.

[16] N. Shankar and S. Owre, “Principles and Pragmatics of Subtyping in PVS.” in *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT ’99*, ser. LNCS, D. Bert, C. Choppy, and P. D. Mosses, Eds., vol. 1827. Springer-Verlag, September 15-18 1999, pp. 37–52.

[17] T. Basten and J. Hooman, “Process Algebra in PVS,” in *TACAS’99*, ser. LNCS, R. Cleaveland, Ed., vol. 1579. Springer-Verlag, 1999, pp. 270–284.

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

[19] B. Dutertre and S. Schneider, “Using a PVS embedding of CSP to verify authentication protocols,” in *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs’97*, ser. LNCS, E. L. Gunter and A. P. Felty, Eds., vol. 1275. Springer-Verlag, 1997, pp. 121–136.

[20] N. Evans and S. A. Schneider, “Verifying security protocols with PVS: widening the rank function approach,” *Journal of Logic and Algebraic Programming*, vol. 64, no. 2, pp. 253–284, August 2005.

[21] A. J. Camilleri, “Mechanizing CSP trace theory in High Order Logic,” *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 993–1004, September 1990.

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

[23] T. F. Melham, “A mechanized theory of the π-calculus in HOL,” *Nordic Journal of Computing*, vol. 1, no. 1, pp. 50–76, 1994.

[24] R. Groenboom, C. Hendriks, I. Polak, J. Terlouw, and J. T. Udding, “Algebraic Proof Assistants in HOL,” in *MPC ’95: Mathematics of Program Construction*, ser. LNCS, vol. 947. Springer-Verlag, 1995, pp. 304–321.

[25] M. B. Josephs and J. T. Udding, “An overview of DI algebra.” in *26th Hawaii Int. Conference on System Science (HICSS 1993)*, T. N. Mudge, V. Milutinovic, and L. Hunter, Eds., vol. I. IEEE Computer Society Press, 1993, pp. 329–338.