Executable Semantics for Compensating CSP

Michael Butler and Shamim Ripon

School of Electronics and Computer Science, University of Southampton, UK,
{mjb,sr03r}@ecs.soton.ac.uk

Abstract. Compensation is an error recovery mechanism for long-running transactions. Compensating CSP is a variant of the CSP process algebra with constructs for orchestration of compensations. We present a simple operational semantics for Compensating CSP and outline an encoding of this semantics in Prolog. This provides a basis for implementation and model checking of the language.

1 Introduction

Web services technology provides a platform on which to develop distributed services. In order to define web service composition, that is, the definition of complex services out of simple ones, *web services choreography* has been introduced. There have been several proposals for describing web services for business processes presented in the recent years including BPML [3] by BPMI, XLANG [20] and BizTalk [16] by Microsoft, WSFL [15] by IBM, BPEL4WS [10] by OASIS (draft standard).

Business transactions involve interaction and coordination between several services. Business transactions need to deal with faults that can arise in any stage of such an environment and this is both difficult and critical. In a long running transaction the usual database approaches, e.g., rollback, are not possible to handle faults. Usually, a long-running transaction interacts with the real world which makes it difficult to undo the transaction. In order to recover from faults in long-running transactions, the concept of compensation was introduced [11]. Compensation is the act of making amendments or making up of a previously completed task. If a long-running transaction fails, appropriate compensations are run to compensate for completed parts of the transaction.

Operational semantics is given by a set of rules which specify how the states of a program change during execution. The overall states of the program are divided into a number of components. Each rule specifies certain precondition on the content of some component and their new content after application of the rule.

The Compensating CSP (cCSP) language was introduced by Butler et al [9] as a language to model long running transactions in the framework of CSP process algebra [12]. The semantics of the cCSP language was described by using denotational semantics (trace semantics). This paper presents the operational semantics of standard as well as compensable processes of compensating CSP using the approach of Plotkin [18]. The operational semantics gives a precise

understanding of the execution of the language. Roscoe [19] describes the operational semantics of standard CSP and our work builds on that.

We make the operational semantics executable by directly encoding the rules in Prolog. Our hope is that this can serve as a useful basis for model checking cCSP processes. XTL [1] is a model checker which allows a wide range of system specification. It accepts specifications written by using high level Prolog predicates describing the transition between different states of the system. Given a Prolog encoding of the operational semantics, the XTL package provides us with an experimental animator and model checker for cCSP.

This paper is organized as follows. Section 2 gives a brief introduction of the cCSP language. Followed by the introduction of the language, Section 3 presents the operational semantics of cCSP. An executable semantics of the operational semantics is presented in Section 4. Section 5 presents the related work and motivates our contribution with respect to them. Concluding remarks are drawn up in Section 6 and some future directions of the present work are mentioned.

2 Compensating CSP

In this section we briefly introduce the cCSP language. The language was inspired by two main ideas: transaction processing features and process algebra, especially CSP. As in CSP, processes in compensating CSP are modelled in terms of atomic events they can engage in and the operators provided by the language support sequencing, choice, parallel composition of processes. In order to support failed transactions, compensation operators are introduced and processes are categorized into standard and compensable processes. We use P, Q to identify standard processes and PP, QQ to identify compensable processes.

The syntax of compensating CSP is summarised in Figure 1. The basic unit of a standard process is an atomic event. Standard process are constructed with the usual CSP operators for choice, sequencing and parallel composition. The process $SKIP$ terminates immediately successfully. The language also provides interrupts and interrupt handling. The primitive process $THROW$ throws an interrupt immediately. In a purely sequential process, the exception causes an immediate disruption to the flow of control. An interrupt handler may be used to catch interrupts: in $P \rhd Q$, an interrupt raised by P triggers execution of the handler Q. In parallel processes, the whole group of parallel processes may fail when one of the processes throws an exception and all the other processes are willing to disrupt their flow of control and yield to the exception. A process that is ready to terminate is also willing to yield to an interrupt. A process may also yield at mid points in its execution. Yield points are inserted in a process though the primitive $YIELD$ process. For example, $P; YIELD; Q$ is willing to yield to an interrupt in between execution of P and Q. Parallel composition is defined so that throwing of an interrupt in one process synchronises with yielding in another process. The current version of cCSP does not support synchronised communication between parallel processes. Parallel process groups synchronise only on joint execution of compensation, joint termination and joint interruption.

Standard processes:

$$
\begin{array}{rll}
P, Q ::= & A & \text{(atomic action)} \\
| & P \ ; \ Q & \text{(sequential composition)} \\
| & P \ \Box \ Q & \text{(choice)} \\
| & P \parallel Q & \text{(parallel composition)} \\
| & SKIP & \text{(normal termination)} \\
| & THROW & \text{(throw an interrupt)} \\
| & YIELD & \text{(yield to an interrupt)} \\
| & P \ \triangleright \ Q & \text{(interrupt handler)} \\
| & [PP] & \text{(transaction block)}
\end{array}
$$

Compensable processes:

$$
\begin{array}{rll}
PP, QQ ::= & P \ \div \ Q & \text{(compensation pair)} \\
| & PP \ ; \ QQ & \\
| & PP \ \Box \ QQ & \\
| & PP \parallel QQ & \\
| & SKIPP & \\
| & THROWW & \\
| & YIELDD &
\end{array}
$$

Fig. 1. Syntax of compensating CSP

A compensable process is one which has compensation actions attached to it. A compensable process consists of a forward behaviour and a compensation behaviour. In the case of an exception, compensation will be executed to compensate the forward behaviour. Both the forward and compensation behaviour are standard processes. The basic way of constructing a compensable process is through the compensation pair construct $P \div Q$, where P is the forward behaviour and Q is its associated compensation. Q should be designed to compensate for the effect of P and may be run long after P has completed.

The parallel and sequential composition operators for compensable processes are designed in a way which ensures that after the failure of a transaction the necessary atomic transactions are performed in an appropriate order to compensate the effect of already performed actions. Sequential composition of compensable processes is defined so that the compensations for all performed actions will be accumulated in the reverse order to their original performance. Parallel composition of compensable processes is defined so that compensations for performed actions will be accumulated in parallel.

By enclosing a compensable process PP in a transaction block $[PP]$ we get a complete transaction which converts the compensable process PP into a standard process. The behaviours of the transaction block are defined in terms of the behaviour of PP. Successfully completed PP represents successful completion of the whole transaction block and compensations are no longer needed. When the forward behaviour of PP throws an interrupt, the compensations are executed in the appropriate order and the interrupt is not observable outside the block.

A standard process can be transformed onto a compensable process by adding to it a compensation process, which actually does nothing (SKIP). The compensable basic processes, which we get from standard basic processes, are as follows:

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

An example of a transaction for processing customer orders in a warehouse is presented in Figure 2 in the cCSP language. The first step in the transaction is a compensation pair. The primary action of this pair is to accept the order and deduct the order quantity from the inventory database. The compensation action simply adds the order quantity back to the total in the inventory database. After an order is received from a customer, the order is packed for shipment, and a courier is booked to deliver the goods to the customer. The *PackOrder* process packs each of the items in the order in parallel. Each *PackItem* activity can be compensated by a corresponding *UnpackItem*. Simultaneously with the packing of the order, a credit check is performed on the customer. The credit check is performed in parallel because it normally succeeds, and in this normal case the company does not wish to delay the order unnecessarily. In the case that a credit check fails, an interrupt is thrown causing the transaction to stop its execution, with the courier possibly having been booked and possibly some of the items having being packed. In case of failure, the semantics of the transaction block will ensure that the appropriate compensation activities will be invoked for those activities that did take place.

$$\begin{aligned}
OrderTransaction &= [ProcessOrder] \\
ProcessOrder &= (AcceptOrder \div RestockOrder)\ ; Fulfill Order \\
Fulfill Order &= BookCourier \div CancelCourier \parallel \\
&\quad PackOrder \parallel \\
&\quad CreditCheck\ ;\ (\quad Ok; SKIPP \\
&\qquad\qquad\qquad \Box\ NotOk\ ; THROWW\ ) \\
PackOrder &= \parallel i \in Items\ \bullet\ (PackItem(i) \div UnpackItem(i))
\end{aligned}$$

Fig. 2. Order transaction example

3 Operational Semantics

The operational semantics is a way of defining the behaviour of processes by specifying atomic transitions on process terms. We will write labelled transition

$$P \xrightarrow{A} P'$$
$$PP \xrightarrow{A} PP'$$

to denote that execution of event A causes the transition from term P or PP to term P' or PP' respectively.

The set of events that a process can perform is called its alphabet. We differentiate between observable and terminal events. The set of observable events is represented by Σ. The terminal events $\Omega = \{\checkmark, !, ?\}$ represent the different ways in which a process may terminate: successful termination is represented by the $\surd$ event, throwing of an interrupt is represented by the ! event and yielding is represented by the ? event. In order to define the semantics we extend the syntax with the null process 0 that cannot perform any events. The terminal events effect standard and compensable processes differently. When a standard process performs a terminal event ω ($\omega \in \Omega$) then the process is finished either normally or abnormally and no further operation occurs.

$$P \xrightarrow{\omega} 0 \qquad (\omega \in \Omega)$$

When a compensable process PP executes a terminal event, instead of evolving to the null process (0), it evolves to a standard process P representing its compensation.

$$PP \xrightarrow{\omega} P \qquad (\omega \in \Omega)$$

In Section 3.2 we will see how these resulting compensations are treated by the various operators for compensable processes.

3.1 Semantics of Standard Processes

This section presents the operational semantics of standard processes of compensating CSP. A process A performs the atomic event and then terminates successfully:

$$A \xrightarrow{A} SKIP \qquad (A \in \Sigma)$$

$SKIP$, $THROW$ and $YIELD$ are primitive processes of cCSP. The effect of terminal events on the special processes are presented here:

$$SKIP \xrightarrow{\checkmark} 0$$

$$THROW \xrightarrow{!} 0$$

$$YIELD \xrightarrow{\checkmark} 0$$

$$YIELD \xrightarrow{?} 0$$

In a sequential composition $P;Q$, P may perform non-terminal events while Q is preserved:

$$\frac{P \xrightarrow{\alpha} P'}{P;Q \xrightarrow{\alpha} P';Q} \qquad (\alpha \in \Sigma)$$

If the first process P terminates normally, then Q starts and the $\surd$ action is hidden from outside:

$$\frac{P \xrightarrow{\surd} 0 \wedge Q \xrightarrow{\alpha} Q'}{P;Q \xrightarrow{\alpha} Q'} \quad (\alpha \in \Sigma \cup \Omega)$$

When the first process P performs a throw or a yield then the whole sequential composition is terminated:

$$\frac{P \xrightarrow{\omega} 0}{P;Q \xrightarrow{\omega} 0} \ (\omega \in \{!, ?\})$$

The interrupt handler is similar to sequential composition, except that the flow of control from the first to the second process is caused by the throw event rather than the $\surd$ event:

$$\frac{P \xrightarrow{\alpha} P'}{P \ \rhd \ Q \xrightarrow{\alpha} P' \ \rhd \ Q} \quad (\alpha \in \Sigma)$$

$$\frac{P \xrightarrow{!} 0 \wedge Q \xrightarrow{\alpha} Q'}{P \ \rhd \ Q \xrightarrow{\alpha} Q'} \quad (\alpha \in \Sigma \cup \Omega)$$

$$\frac{P \xrightarrow{\omega} 0}{P \ \rhd \ Q \xrightarrow{\omega} 0} \quad (\omega \in \Omega \wedge \omega \neq !)$$

In choice operation occurrence of an event in either of the processes resolves the choice:

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

We are only considering the parallel processes synchronising on terminal events. In a parallel composition, either process may progress independently by performing a non-terminal event:

$$\frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'} \quad (\alpha \in \Sigma)$$

Processes placed in parallel will synchronise on joint termination or joint interruption. If we consider ω and ω' are the terminal events of two distinct parallel processes then their joint event will be $\omega\&\omega'$. The definition of this operator is shown in Table 1. Synchronisation of standard processes is defined as follows:

$$\frac{P \xrightarrow{\omega} 0 \ \wedge \ Q \xrightarrow{\omega'} 0}{P \parallel Q \xrightarrow{\omega\&\omega'} 0}$$

ω	ω'	$\omega \& \omega'$
!	!	!
!	?	!
!	$\surd$	!
?	?	?
?	$\surd$	?
$\surd$	$\surd$	$\surd$

Table 1. Synchronization of terminal events

3.2 Semantics of Compensable Processes

In this section we present the semantics of the operators for compensable processes. Recall that a compensable process consists of forward behaviour and compensation behaviour.

The compensation pair $(P \div Q)$ is constructed from two standard processes. The first one is called forward process which is executed during normal execution and the second one is called the compensation of the forward process which is stored for future use when it is required for compensation. If the forward process can perform a non-terminal event, then so can the pair:

$$\frac{P \xrightarrow{\alpha} P'}{P \div Q \xrightarrow{\alpha} P' \div Q} \quad (\alpha \in \Sigma)$$

If the forward process terminates normally, then the pair terminates with Q as the resulting compensation.

$$\frac{P \xrightarrow{\surd} 0}{P \div Q \xrightarrow{\surd} Q}$$

If the forward process terminates abnormally, then so does the pair, resulting in an empty compensation process:

$$\frac{P \xrightarrow{\omega} 0}{P \div Q \xrightarrow{\omega} SKIP} \ (\omega \in \{!, ?\})$$

The definition of the compensation pair defined in the traces model of cCSP [9] has a subtle difference to that presented here. An extra behaviour for the compensation pair was included in the traces model definition which allows the compensation pair to yield immediately with an empty compensation. This forces an automatic yield at the beginning of the compensation pair. The same behaviour can be obtained using the definition presented here by adding a yield sequentially followed by the forward process.

$$P \div' Q \mathrel{\hat{=}} (YIELD; P) \div Q$$

As for the standard case, in a sequential composition $PP; QQ$, PP may perform non-terminal events while QQ is preserved:

$$\frac{PP \xrightarrow{\alpha} PP'}{PP; QQ \xrightarrow{\alpha} PP'; QQ} \quad (\alpha \in \Sigma)$$

If PP throws or yields to an interrupt, the whole process terminates and the compensation from PP is returned:

$$\frac{PP \xrightarrow{\omega} P}{PP; QQ \xrightarrow{\omega} P} \quad (\omega \in \Omega \wedge \omega \neq \surd)$$

If PP terminates normally, QQ commences and the compensation from PP should be maintained to be composed with the compensation from QQ at a later stage. In order to deal with this we introduce a new auxiliary construct to the language of the form $\langle QQ, P\rangle$. The effect of $\langle QQ, P\rangle$ is to execute the forward behaviour of QQ and then compose the compensation from QQ with P. This is used to define the transfer of control in a sequential composition:

$$\frac{PP \xrightarrow{\surd} P \wedge QQ \xrightarrow{\alpha} QQ'}{PP; QQ \xrightarrow{\alpha} \langle QQ', P\rangle} \quad (\alpha \in \Sigma)$$

However, if QQ involves in a terminal event after PP terminates normally, then instead of introducing the new auxiliary construct, the maintained compensations of both processes are accumulated.

$$\frac{PP \xrightarrow{\surd} P \wedge QQ \xrightarrow{\omega} Q}{PP; QQ \xrightarrow{\omega} Q \ ; \ P} \quad (\omega \in \Omega)$$

The process QQ in the construct $\langle QQ, P\rangle$ can perform non-terminating events:

$$\frac{QQ \xrightarrow{\alpha} QQ'}{\langle QQ, P\rangle \xrightarrow{\alpha} \langle QQ', P\rangle} \quad (\alpha \in \Sigma)$$

When QQ terminates then its compensation is composed in front of the existing compensation, which ensures that the compensations are accumulated in reverse order to their original sequential operation:

$$\frac{QQ \xrightarrow{\omega} Q}{\langle QQ, P\rangle \xrightarrow{\omega} Q; P} \ (\omega \in \Omega)$$

An event in PP or QQ resolves the choice in a choice composition.

$$\frac{PP \xrightarrow{\alpha} PP'}{PP \ \Box \ QQ \xrightarrow{\alpha} PP'} \qquad \frac{QQ \xrightarrow{\alpha} QQ'}{PP \ \Box \ QQ \xrightarrow{\alpha} QQ'} \quad (\alpha \in \Sigma)$$

The terminal events ($\surd$,!,?) also resolve the choice resulting in the corresponding compensations:

$$\frac{PP \xrightarrow{\omega} P}{PP \,\Box\, QQ \xrightarrow{\omega} P} \qquad \frac{QQ \xrightarrow{\omega} Q}{PP \,\Box\, QQ \xrightarrow{\omega} Q} \quad (\omega \in \Omega)$$

Parallel processes evolve independently through non-terminal events:

$$\frac{PP \xrightarrow{\alpha} PP'}{PP \parallel QQ \xrightarrow{\alpha} PP' \parallel QQ} \qquad \frac{QQ \xrightarrow{\alpha} QQ'}{PP \parallel QQ \xrightarrow{\alpha} PP \parallel QQ'} \quad (\alpha \in \Sigma\})$$

As the processes are compensable, when they synchronise over any terminal events, the forward processes are terminated and the corresponding compensation processes will be accumulated in parallel:

$$\frac{PP \xrightarrow{\omega} P \;\wedge\; QQ \xrightarrow{\omega'} Q}{PP \parallel QQ \xrightarrow{\omega \& \omega'} P \parallel Q}$$

Although a transaction block is a standard process rather than a compensable process, we describe its semantics in this section rather than the previous one since it requires an understanding of the semantics of compensable processes. A transaction block is formed from a compensable process PP by enclosing PP in a transaction block $[PP]$. A transaction block converts a compensable process into a standard process. A non-terminal event changes the state of the process inside the block:

$$\frac{PP \xrightarrow{\alpha} PP'}{[PP] \xrightarrow{\alpha} [PP']} \quad (\alpha \in \Sigma)$$

Successful completion of the forward behaviour of the compensable process of a transaction block represents successful completion of the whole block and compensation is no longer needed and it is discarded:

$$\frac{PP \xrightarrow{\surd} P}{[PP] \xrightarrow{\surd} 0}$$

When the forward behaviour throws an exception, then the resulting compensation is run:

$$\frac{PP \xrightarrow{!} P \wedge P \xrightarrow{\alpha} P'}{[PP] \xrightarrow{\alpha} P'} \qquad (\alpha \in \Sigma \cup \Omega)$$

Since a transaction block is a standard process, P' in this rule is not a compensation that is stored for later execution, rather it describes the behaviour of $[PP]$ after execution of event α.

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

3.3 Correspondence with Trace Semantics

When both an operational and denotational semantics are defined for a particular language, a natural question is how these are related. In this section, we briefly describe the way in which we are attempting to show the correspondence between the operational semantics presented in this paper and the denotational semantics presented in traces model shown in [9].

Given our operational rules for cCSP which defines a labelled transition relation between process terms, we can define a lifted transition relation labelled by sequences of events in the usual way:

$$P \xrightarrow{s} Q$$

Roscoe [19] describes how to extract the traces from operational rules as follows:

$$traces(P) = \{s \in \Sigma^{*\checkmark} \mid \exists Q.P \xrightarrow{s} Q\}$$

We derive traces from the operational rules in a similar way. In the standard traces model for CSP, process are modelled as prefixed-closed sets of traces. However, in the traces model for cCSP, processes are modelled as sets of completed traces, where a completed trace ends in one of the terminal symbols $\Omega = \{\checkmark, ?, !\}$. The traces model for cCSP is not closed under trace prefixes.

Standard traces are defined as set of traces of the form $p\langle\omega\rangle$ where $p \in \Sigma^*$ and $\omega \in \Omega$. The derived traces of a standard cCSP process P are denoted by $DT(P)$ which is defined as follows:

$$DT(P) = \{\ p\langle\omega\rangle \mid P \xrightarrow{p\langle\omega\rangle} 0\ \}$$

As compensable processes contain forward behaviour and compensation behaviour, they are modelled as pairs of traces of the form $(p\langle\omega\rangle, p'\langle\omega'\rangle)$ where $p\langle\omega\rangle$ represents forward behaviour and $p'\langle\omega'\rangle$ represents the corresponding compensation behaviour. The derived traces of a compensable cCSP process PP are denoted by $DT(PP)$ which is defined as follows:

$$DT(PP) = \{\ (p\langle\omega\rangle, p'\langle\omega'\rangle) \mid \exists P \cdot\ PP \xrightarrow{p\langle\omega\rangle} P \wedge P \xrightarrow{p'\langle\omega'\rangle} 0\ \}$$

Let $T(P)$ be the traces of a standard term P as defined in [9]. Similarly for $T(PP)$. By structural induction over process terms P and PP, it should be possible to prove the following correspondence:

$$DT(P) = T(P)$$
$$DT(PP) = T(PP)$$

4 Prolog Implementation

In this section we outline a prolog implementation of the operational semantics presented in Section 3. We encode the operational rules as Prolog clauses and we

use a tool which can animate this encoded semantics and support model checking and refinement of the specification. XTL [1] is a model checker which allows a wide range of system specification. It accepts specifications written by using high level Prolog predicates describing the transition between different states of the system. The XTL animator supports step by step animation showing transition between different states of specification and also support backtracking.

The input language for XTL is very simple. There are two key predicates that can be entered into XTL: `trans/3` and `prop/2` where:

`trans(A,S1,S2)`: A transition from state S1 to state S2 by the action A.
`prop(S,P)`: property P holds in state S.

Consider the following simple system specified in this way:

```
trans(a1,p,q). trans(a2,q,p). trans(a3,r,r).
prop(p,safe). prop(q,safe). prop(r,unsafe).
```

These lines specify that by the action `a1`, there is a transition from `p` to `q`, that action `a2` causes the reverse transition and action *a3* causes `r` to `r`. The property clauses specify that state `p` and `q` are safe and that `r` is unsafe. The XTL model checker supports checking of temporal properties written in CTL (Computation Tree Logic) of systems specified in this way.

As the operational semantics of compensating CSP are described by using operational rules, they are easily transferable to corresponding `trans/3` predicates. We reproduce some operational rules and their corresponding Prolog predicates. For example, consider one of the rules for sequential composition of standard processes:

$$\frac{P \xrightarrow{\alpha} P'}{P;Q \xrightarrow{\alpha} P';Q} \qquad (\alpha \in \Sigma)$$

The Prolog representation of this is:

```
trans(seq(P,Q),A,seq(P1,Q)):-
        member(A,sigma),
        trans(P,A,P1).
```

Compensable processes are encoded in a similar way with the compensable operators being differentiated from the standard ones. For example, consider the following rule for compensable sequential composition:

$$\frac{PP \xrightarrow{\alpha} PP'}{PP;QQ \xrightarrow{\alpha} PP';QQ} \quad (\alpha \in \Sigma)$$

This is represented in prolog as:

```
trans(cseq(PP,QQ),A,cseq(PP1,QQ)):-
        member(A,sigma),
        trans(PP,A,PP1).
```

The XTL package provides us with an experimental animator and model checker for cCSP. We are currently investigating the use of this further. We are also investigating the use of the prolog encoding as a basis for a refinement checking tool. Refinement checking is currently supported by the ProB model checker [14] using similar prolog techniques to XTL.

5 Related Work

Bocchi et al [2] define a language πt-calculus for modelling long-running transactions based on Milner's π-calculus [17]. The πt-calculus includes a transaction construct that contains a compensation handler and a fault manager. In this approach a transaction process remains active as long as its compensation might be required. This doesn't allow for the sequential composition of compensable transactions in which compensations are composed in reverse order.

Recently, Laneve and Zavattaro [13] defined a calculus for web transactions called webπ which is an extension of asynchronous π-calculus with timed transaction construct. The major aspects considered in webπ are that the processes are interruptible, failure handlers are activated when main processes are interrupted and time which is considered in order to deal with latency of web activities or with message losses. A transaction executes either until its termination or until it fails and upon failure the compensation is activated. However, it has the similar problem as πt-calculus where compensations of sequentially composed transactions are not preserved in reverse order and it is not possible to get the compensation of a successfully completed process after the failure of a process composed sequentially with the previous one.

One of the authors (Butler) was involved in the development of the StAC (Structured Activity Compensation) language [6, 7] for modelling long-running business transactions which includes compensation constructs. An important difference between StAC and cCSP is that instead of the execution of compensations being part of the definition of a transaction block, StAC has explicit primitives for running or discarding installed compensations (*reverse* and *accept* respectively). This separation of the *accept* and *reverse* operators from compensation scoping prevents the definition of a simple compositional semantics: the semantics of the reverse operator cannot be defined on its own as its behaviour depends on the context in which it is called. This necessitated the use of configurations involving installed compensation contexts in the operational semantics for StAC. Note that BPEL also has an operator for explicit invocation of compensation. A mapping from BPEL to StAC may be found in [8].

Bruni et al [5] have developed an operational semantics for a language with similar operators to cCSP, including compensation pairs and transaction blocks (or sagas as they call them). As in cCSP, and unlike StAC, the invocation of compensation in a saga is automatic depending on failure or success which leads to a neater operational semantics. However, unlike the work presented here, the operational semantics in [5] is defined by using big-step semantics. Big-step semantics describe how the overall results of the execution are obtained. The big

step semantics are closer to the trace semantics while our small-step semantics describes how compensating processes should be executed. A comparison of the operators of cCSP and the language described in [5] may be found in [4].

6 Conclusions and Future Work

Compensating CSP has evolved from the development of the StAC language. StAC has a somewhat complicated operational semantics because of the need to maintain compensation contexts in process configurations. Compensating CSP was developed through a trace semantics which forces a compositional semantic definition. This leads to a more structured treatment of compensation which in turn has lead to a much simpler operational semantics than that of StAC. We are currently working on proving the corespondence between the trace and operational semantic models of cCSP.

Our operational semantics provides the basis for a prototype model checker for cCSP as well as a basis for an implementation strategy for a language with compensations.

7 Acknowledgements

Thanks to Hernan Melgratti and to the anonymous WS-FM05 referees for useful comments on an earlier version of the paper. Thanks for Michael Leuschel for help with XTL.

References

1. Juan C. Augusto, Michael Leuschel, Michael Butler, and Carla Ferreira. Using the extensible model checker XTL to verify StAC business specifications. In *3rd Workshop on Automated Verification of Critical Systems (AVoCS 2003)*, pages 253–266, Southampton, UK, 2003.
2. Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calulus for long-running transactions. In *FMOODS'03*, volume 2884 of *LNCS*, pages 124–138. Springer-Verlag, 2003.
3. Business Process Modeling Language (BPML). [www.bpmi.org].
4. Roberto Bruni, Michael Butler, Carla Ferreira, Tony Hoare, Hernan Melgratti, and Ugo Montanari. Reconciling two approaches to compensable flow composition. Technical report, 2005.
5. Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL*, pages 209–220, 2005.
6. Michael Butler and Carla Ferreira. A process compensation language. In *Integrated Formal Methods(IFM'2000)*, volume 1945 of *LNCS*, pages 61 – 76. Springer-Verlag, 2000.
7. Michael Butler and Carla Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Coordination 2004*, volume 2949 of *LNCS*. Springer-Verlag, 2004.

8. Michael Butler, Carla Ferreira, and M.Y. Ng. Precise modelling of compensating business transactions and its application to BPEL. *Journal of Universal Computer Science, to appear*, 2005.
9. Michael Butler, Tony Hoare, and Carla Ferreira. A trace semactics for long-running transaction. In A.E. Abdallah, C.B. Jones, and J.E. Sanders, editors, *Proceedings of 25 Years of CSP*, volume 3525 of *Springer LNCS*, London, 2004.
10. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. *Business process execution language for web services, version 1.1.*, 2003. [http://www-106.ibm.com/developerworks/library/ws-bpel/].
11. H. Garcia-Molina and K. Salem. Sagas. In *ACM SIGMOD*, pages 249–259. ACM Press, 1987.
12. C.A.R. Hoare. *Communicating Sequential Process*. Prentice Hall, 1985.
13. Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *LNCS*, pages 282–298, 2005.
14. Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, September 2003.
15. Frank Leymann. The web services flow language (WSFL 1.0). Technical report, Member IBM Academy of Technology, IBM Software Group, 2001. [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf].
16. B. Metha, M. Levy, G. Meredith, T. Andrews, B. Beckman, J. Klein, and A. Mital. Biztalk server 2000 business process orchestration. *IEEE Data Engineering Bulletin,*, 24(1):35–39, 2001.
17. Robin Milner. A calculus of mobile processes. *Journal of Information and computing*, 100(1):1–77, 1992.
18. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, September 1981.
19. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, pearson edition, 1998.
20. S.Thatte. *XLANG: Web Services for Business Process Design*. Microsoft Corporation, 2001. [www.gotdotnet.com/team/xml/wsspace/xlang-c].