A Process Compensation Language

Michael Butler and Carla Ferreira

Declarative Systems and Software Engineering Group (DSSE)
Dept. of Electronics and Computer Science, Univ. of Southampton,
Highfield, Southampton SO17 1BJ, United Kingdom
{mjb,cf}@ecs.soton.ac.uk

Abstract. This paper presents a formal language for the design of component-based enterprise system. The language (StAC) allows the usual parallel and sequential behaviours, but most significant is the concept of compensation that allows a previous action to be undone. The semantics of the language is given by an operational approach. The specification of a system is composed by a set of StAC processes that describe the behaviour of the system and a set of B operations that describe basic computations. Operational semantics is used to justified the integration of StAC processes with B operations.

1 Introduction

The work presented in this paper is the result of the collaboration of the DSSE Group with the Transaction Processing Design and New Technology Development Group at IBM UK Laboratories, Hursley. The collaboration has been concerned with approaches and techniques for component-based enterprise system development.

So far, with support from IBM UK Laboratories, we have defined a formal design language suitable for a heterogeneous distributed environment. IBM imposed some particular features on the language:

– The target enterprise solution should be built by stitching together Enterprise Java Beans [7] (EJBs).
– The language should allow sequential and parallel composition of behaviours.
– It should be possible for earlier actions to be undone, which is called "compensation", whereby the system keeps track of the compensations that need to be executed if part of a process is to be aborted.

The IBM group believes that compensation gives more flexibility than the traditional commit-rollback approach to transaction processing. This flexibility is necessary for the heterogeneous distributed environment on which modern enterprise systems operate. Instead of restoring the system to the state before activities where performed, activities can have a compensation activity associated with them.

The main goal of our collaboration with IBM was to define the semantics of compensation, which is a complex concept. The complexity arises in particular because of the combination of compensation with parallel execution.

In this paper we define a formal language for the design of heterogeneous distributed systems, which we called StAC (**St**ructured **A**ctivity **C**ompensation) language. In this approach the description of a system determines the way to "connect" simple components in order to create a complete system. In the StAC language the components are B [1] operations. B is a model-oriented formal notation. We use B instead of EJBs because we want the language to have a formal semantics.

A system specification has two components, the StAC that describes the execution order of the operations, and a B specification that describes the state of the system and also its basic operations. Since there are commercial tools available to support the development in B, the B specification can be animated and also proof obligations can be generated and proved.

An operational semantics is defined for the StAC language and its integration with B is also formalised through an operational approach.

In Section 2 we present informally the StAC language and specify in StAC an e-bookstore, in Section 3 we complete the example specification by describing the B operations. Section 4 describes the semantics of StAC, and in the last section, we discuss the integration of StAC with B.

2 The StAC Language

We can say informally that in StAC a system is specified as a process, and such a process will be decomposed into several sub-processes in a top-down approach. At the bottom level there will only be activities (each activity is a simple computation), so they can not be further decomposed. Formally a system is described by a set of equations of the form

$$N = P,$$

where N is a process identifier and P is an expression that can contain several process identifiers, including N, since the equations can have recursion. We have determined, for simplicity reasons, that the first equation describes the overall system being specified.

The distinctive characteristic about the StAC language is the concept of compensation. This concept can have many interpretations, the most common being recovering from an error. Each process can have a corresponding compensation process, so executing a process has two consequences: the execution of the process itself, and its compensation process must be preserved by the system[1].

Next we present two informal definitions of the concepts of compensation process and compensate, later on this two concepts will be formally defined.

Definition 1 (Compensation Process). *Process representing activities to be performed to compensate some behaviour.*

Definition 2 (Compensate). *Action of invoking the compensation process.*

[1] Later on we will define what the expression "preserved by the system" exactly means.

The StAC language has some usual process combinators, like sequence and parallel. Besides these it has specific combinators to deal with the compensation. Next we present the definition of the StAC in Backus-Naur form:

$$
\begin{array}{rll}
\text{Process} ::= & A & \text{(activity)} \\
 & \mid 0 & \text{(skip)} \\
 & \mid b \rightarrow P & \text{(condition)} \\
 & \mid rec(N) & \text{(recursion)} \\
 & \mid P; Q & \text{(sequence)} \\
 & \mid P \parallel Q & \text{(parallel)} \\
 & \mid \parallel_{x \in X} . P_x & \text{(generalised parallel)} \\
 & \mid P [] Q & \text{(choice)} \\
 & \mid []_{x \in X} . P_x & \text{(generalised choice)} \\
 & \mid P \div Q & \text{(compensation pair)} \\
 & \mid \boxtimes & \text{(compensate)} \\
 & \mid \boxdot & \text{(commit)} \\
 & \mid [P] & \text{(compensation scoping)}
\end{array}
$$

An activity A represents an atomic change in the state: if Σ is the set of all possible states, then A is a relation on Σ. In Section 3, we show how the B notation is used to describe state and activities. In the rest of paper we consider the capital letters A to D are activities, and the letters P and Q are processes.

In the conditional operator process, P is guarded by a boolean function b. This boolean function can consult the state, i.e., $b : \Sigma \rightarrow Bool$. The recursive operator $rec(N)$ enables the use of the process identifier N of the right-side of the equation to be used in the left-side term of the equation. The sequence $P; Q$ forces an order in the execution of P and Q, P must be executed first and only when P finishes can Q be executed. In parallel process $P \parallel Q$, the execution of the activities of P and Q is interleaved. Generalised parallel extends the parallel operator over a set X, which can either be finite or infinite. The choice $P[]Q$ behaves as P or Q and generalised choice just extends that behaviour over a set. In this language the choice between several processes is made by the environment. Notice that the $[]$ operator causes non-determinism in some cases, if we consider the following example:

$$(A; B) [] (A; C)$$

when activity A occurs its not possible to determine which one of the two behaviours $A; B$ or $A; C$ will be chosen. In this case, the choice is made by the system rather than the environment.

The next set of operators is related to the compensation concept. The compensation pair $P \div Q$ expresses that P has Q as its compensation process. The compensation process is constructed in the reverse order of process execution, for example:

$$(A \div A'); (B \div B') \tag{1}$$

behaves as $A; B$ and has the compensation process $B'; A'$. A compensation process can be viewed as a stack where processes are pushed into the top of the stack.

The compensate operator $\boxtimes$ causes the compensation process to be executed. Consider the process (1) followed by $\boxtimes$:

$$(A \div A'); (B \div B'); \boxtimes. \tag{2}$$

As we saw before, process (1) behaves as $A; B$, and then $\boxtimes$ operator causes the compensation process to be executed, so the overall behaviour of process (2) is $(A; B); (B'; A')$ which we write as $A; B; B'; A'$.

The commit operator $\boxdot$ clears the compensation process, meaning that after a commit the compensation process is set to 0. Consider again process (1). If now we append to it the $\boxdot$ operator we have:

$$(A \div A'); (B \div B'); \boxdot.$$

As we already saw the compensation process of process (1) is $B'; A'$, but after the $\boxdot$ operator the compensation process is 0. Another example,

$$(A \div A'); (B \div B'); \boxdot; \boxtimes$$

behaves as $A; B$ because when the $\boxtimes$ operator is called the compensation process $B'; A'$ has already been cleared by the $\boxdot$ operator.

As we mentioned before the complexity of StAC language arises from the combination of compensation with parallel execution. Given the following parallel processes:

$$(A \div A') \parallel (B \div B')$$

the execution of processes $A \div A'$ and $B \div B'$ is interleaved, implying that the execution of their compensation process should also be interleaved, so the resulting compensation process is $A' \parallel B'$.

Next we will consider the combination of compensation with choice, the process:

$$(A \div A') \,[]\, (B \div B')$$

behaves as A or B and the compensation process in the first case is A' and in the second case is B'.

The compensation scoping operator $[P]$ creates a new compensation process within the square brackets. In the following process:

$$(A \div A'); [\,(B \div B'); (C \div C')\,],$$

the compensation process within the brackets is just $C'; B'$, and A' is excluded because its outside the brackets. If we added the compensate operator as follows:

$$(A \div A'); [\,(B \div B'); (C \div C'); \boxtimes\,],$$

the overall process would behave as $A; B; C; C'; B'$, since the $\boxtimes$ just executes the compensation process within the brackets. Also, the process:

$$(A \div A'); [\,(B \div B'); (C \div C'); \boxdot\,]; (D \div D')$$

has $D'; A'$ as compensation. Since the commit operator is inside the brackets, it just clears the compensation process $(C'; B')$ that is within the brackets. Another feature of the compensation scoping operator is that compensation is remembered if compensate is not performed, as in the example:

$$(A \div A'); [\,(B \div B'); (C \div C')\,]; (D \div D').$$

Here, the compensation process is $D'; C'; B'; A'$, which includes the compensation process $C'; B'$ defined inside the brackets. $C'; B'$ is retained because there is no commit with the brackets.

The StAC language permits nested compensation pairs as in the next example:

$$A \div (B \div C). \tag{3}$$

The process (3) behaves as A and has the compensation process $B \div C$. Lets review what happens when the compensate operator is appended to process (3),

$$(A \div (B \div C)); \boxtimes. \tag{4}$$

First A is executed, after the $\boxtimes$ operator invokes the compensation process $B \div C$, so the behaviour of process (4) is $A; B$ with compensation process C.

Example: E-Bookstore

The E-Bookstore is a typical example of an e-business. In this example each client defines a limited budget and has an e-basket where the selected books are kept. Every time the client selects a book, the budget is checked to see if it was exceeded, in this case the book is returned to the e-shelf. When the client finishes shopping he can either pay or abandon the bookstore, in the later case all selected books have to be returned to the shelf. Next, we present the StAC specification of the e-bookstore:

$$\begin{array}{ll}
\mathit{Bookstore} & = \|_{\,c \in C}.\mathit{Client}(c) \\
\mathit{Client}(c) & = \mathbf{Arrive}(c); \mathit{ChooseBooks}(c); ((\mathbf{Quit}(c); \boxtimes)\ []\ \mathit{Pay}(c)); \boxdot; \mathbf{Exit}(c) \\
\mathit{ChooseBooks}(c) & = \mathbf{Checkout}(c)\ []\ (\mathit{ChooseBook}(c); \mathit{ChooseBooks}(c)) \\
\mathit{ChooseBook}(c) & = []_{b \in B}.[(\mathbf{AddBook}(c,b) \div \mathbf{ReturnBook}(c,b)); \mathbf{overBudget}(c) \rightarrow \boxtimes] \\
\mathit{Pay}(c) & = \mathbf{ProcessCard}(c); \neg\mathbf{accepted}(c) \rightarrow \boxtimes
\end{array}$$

Notice that some processes are written with a different font, e.g., **AddBook**, this means that those processes are activities, and as mentioned before they can not be further decomposed. Activities will be specified as B operations in the next section.

In the above specification the bookstore is represented by a infinite set of parallel *Client* processes. This implies that the process *Bookstore* never ends, so it is always available and if a real client exits the bookstore he can later return with a new *Client* process. Each *Client* has a thread of compensation independent from all the other *Client* parallel processes. Also, each *Client* is a sequence of five processes. The first one is *Arrive* that creates and initialises the client information, setting the budget to a value determined by the client. The next process is *ChooseBooks*, followed by a choice between paying the books in the basket or abandon the bookstore without buying any books. If the client chooses to quit, the process $\boxtimes$ is invoked causing the return of all books in the client's basket to the shelf. The fourth process is $\boxdot$ which discharges all compensation information. The last process is *Exit* that removes the client from the bookstore, clearing all the information related to that client. The process *ChooseBooks* is a recursive process where the client chooses between selecting individual books (process *ChooseBook*) and returning afterwards to *ChooseBooks* process or stop selecting books, choosing **Checkout**(**c**). *ChooseBook* creates a new thread of compensation, using scoping brackets. This new thread is only related to one book. Within *ChooseBook* there is a compensation pair, **AddBook** compensated by **ReturnBook**, and the compensation is executed only if adding that book to the basket exceeds the budget. In this case executing the compensation implies returning the book that has just been added to the basket rather than all books in the basket. In the process *Pay*, the clients card is processed, and if the card is rejected, the compensation is executed, so all selected books are returned to the shelf.

3 Describing Activities in B

B AMN is a model-oriented formal notation and is part of the B-method developed by Abrial. In the B-method, a system is defined as an abstract machine consisting of some state and some operations acting on the state.

MACHINE *Bookstore*
VARIABLES v
INVARIANT I
OPERATIONS
$\quad\ldots$
END

Fig. 1. *Bookstore* abstract machine

The bookstore abstract machine has the structure outlined in Figure 1. The abstract machine consists of some variables which are described using set-theoretic constructs. The invariant is a set of first-order predicates. Operations act on the variables while preserving the invariant and can have input and output parameters. Initialisation and operations are written in the generalised substitution notation of B AMN which includes constructs such as assignment, guarded statements and unbounded choice.

Machine State

Next we describe the machine state of our example, which is defined by the aggregation of the clauses VARIABLES and INVARIANT:

$$
\begin{array}{l}
\text{MACHINE} \quad \mathit{Bookstore} \\
\text{VARIABLES} \quad \mathbf{budget}, \mathbf{basket}, \mathbf{price}, \mathbf{accepted} \\
\text{INVARIANT} \\
\quad \mathbf{budget} \in C \rightarrow \mathbb{N} \ \wedge \\
\quad \mathbf{basket} \in C \rightarrow \mathbb{F}(B) \ \wedge \\
\quad \mathbf{price} \in B \rightarrow \mathbb{N}_1 \\
\quad \mathbf{accepted} \in C \rightarrow BOOL \ \wedge \\
\quad \ldots \\
\text{END}
\end{array}
$$

The clause VARIABLES names the variables of the abstract machine such as **budget**, **basket**, **price** and **accepted**. In the INVARIANT part we specify the types of the variables introduced in the previous clause. The variable **budget** is a function that for each client returns the maximum amount of money the client intends to spend in the bookstore. Variable **basket** is also a function but in this case it returns the books selected by each client. The **price** function contains the price of each book, this variable is necessary in order to verify if a client has exceed his budget. The variable **accepted** is a function that for each client determines if the clients card was accepted or rejected.

Machine Operations

The operations defined in the B specification are the activities of the process *Bookstore* described in Section 2. The activities are **Arrive**, **Checkout**, **AddBook**, **ReturnBook**, **Quit**, **ProcessCard** and **Exit**.

We will describe in detail some operations, the first one is the operation **AddBook**,

$$
\begin{array}{ll}
\mathbf{AddBook}(c, b) \mathrel{\hat{=}} & \text{SELECT} \ \ c \in C \ \wedge \ b \in B \ \wedge \ b \notin basket(c) \ \ \text{THEN} \\
& \qquad basket(c) := basket(c) \cup \{b\} \\
& \text{END}
\end{array}
$$

The SELECT construct enables the operation only if all conditions are true, in this case c must be in the set C of clients, b must be a book of the set B, and also the book b must not be already in the basket of the client. The last condition was just added for simplicity reasons, we could extend the **basket** variable to return a bag instead a set of books. If all conditions are met book b is added to the basket of client c. The operations **ReturnBook**, **Arrive** and **Exit** are similar to the operation described above in the sense that they are enabled by certain conditions and all of them cause a change in the machine state.

Next we describe the operation **Checkout**:

$$\mathbf{Checkout}(c) \mathrel{\hat{=}} \text{SELECT} \ \ c \in C \ \ \text{THEN} \ \ \mathit{skip} \ \ \text{END}$$

This operation is enable if client c is in the set C. Since **Checkout** is used in the StAC process *Bookstore* to exit the recursive definition of *ChooseBooks*, this operation does not need to perform any explicit action. Operation **Quit** is similar to operation **Checkout**, and is used to determine which action the client wants to perform, quit the bookstore or pay the books.

The boolean function **overBudget** is specified as a B definition, since it only consults the machine state:

$$\mathbf{overBudget}(c) \overset{def}{=} budget(c) \geq \sum b \, . \, (b \in \mathbf{basket}(c) \mid \mathbf{price}(b))$$

This boolean functions calculates if the total price of the books in the client's basket exceeds the initial budget.

Operation **ProcessCard** sets the variable **accepted**. The variable **accepted** is used in the process *Pay* to trigger the compensation process when the card is rejected.

$$\begin{array}{l} \mathbf{ProcessCard}(c) \mathrel{\hat{=}} \text{SELECT } \ c \in C \ \text{ THEN} \\ \qquad\qquad\qquad\qquad \text{CHOICE } \ accepted := TRUE \text{ OR } accepted := FALSE \\ \qquad\qquad\qquad\quad\ \text{END} \end{array}$$

Operation **ProcessCard** is described as a choice between attributing the value $TRUE$ or $FALSE$ to the variable **accepted** depending on the card being accepted or rejected. The need for this variable is due to the fact that conditional processes must only use boolean functions has guards, and **ProcessCard** is not a boolean function, because for the same state it can assign different values to variable **accepted**.

We can state that any guards of conditional processes should be specified in B as boolean expressions of the variables of the state machine.

4 The StAC$_i$ Language

In this section we introduce the StAC$_i$ (**St**ructured **A**ctivity **C**ompensation with **i**ndexes) language which extends the StAC language by adding different threads of compensation to a process. The main reasons for creating this new language is that StAC$_i$ has a clear semantics for compensation and makes it easier to describe parallel compensation.

In the next sections we define formally the syntax and the semantics of StAC$_i$ language. This new language will be used to define the semantics of StAC. To achieve that we constructed a translation function from StAC to StAC$_i$ terms.

4.1 Abstract Syntax

The StAC$_i$ language extends the concept of compensation present in StAC language. In StAC$_i$ a process can have several compensation threads, each one with a independent compensation process. The new operators reflect the extension of the StAC language. Operator $P \div_i Q$ denotes that Q is the compensation process of P within the thread i. In same way operators $\boxtimes_i$ and $\boxdot_i$ compensate and commit, not the general compensation process, but the compensation process of the thread i. The new operator $J \rhd i$ merges all compensation processes of the threads belonging to J into the compensation process of thread i.

A process in the StAC_i language is defined by the following Backus-Naur form:

$$
\begin{array}{rll}
\text{Process} ::= & A & \text{(activity)} \\
& \mid 0 & \text{(skip)} \\
& \mid b \rightarrow P & \text{(condition)} \\
& \mid rec(N) & \text{(recursion)} \\
& \mid P; Q & \text{(sequence)} \\
& \mid P \parallel Q & \text{(parallel)} \\
& \mid \parallel_{x \in X}.P_x & \text{(generalised parallel)} \\
& \mid P[]Q & \text{(choice)} \\
& \mid []_{x \in X}.P_x & \text{(generalised choice)} \\
& \mid P \div_i Q & \text{(compensation pair)} \\
& \mid \boxtimes_i & \text{(compensate)} \\
& \mid \boxdot_i & \text{(commit)} \\
& \mid J \rhd k & \text{(merge)}
\end{array}
$$

As expected both StAC and StAC_i languages are very similar, StAC_i has some additional symbols i, k, J and $\rhd$. Symbols i and k are are members of the set of indexes I, also J is a subset of I. $J \rhd k$ is a new operator and it will be described formally in the next section.

5 Operational Semantics

In this section we present the operational semantics for the StAC language. The main purpose in defining the operational semantics was the formalisation of the concepts present in the StAC language, which was the aim of the collaboration with IBM. Also, we will justify (section 6) the integration of StAC and B through an operational approach.

The semantic domain of an StAC_i program is a tuple,

$$(P, C, \sigma) \;\in\; Process \times (I \rightarrow Process) \times \Sigma$$

which we call a *Configuration*. In the above tuple, C is the set of compensation threads for the process P, such that for each index i, $C(i)$ represents the compensation process of thread i. Σ represents the state of the B machine. We will write a labelled transition

$$(P,\, C,\, \sigma) \xrightarrow{A} (P',\, C',\, \sigma')$$

to denote that executing activity A may cause a configuration transition from $(P,\, C,\, \sigma)$ to $(P',\, C',\, \sigma')$.

In any configuration, the choice between enabled transitions with different labels is made by the environment while the choice between enabled transitions with the same label is made by the system.

Some rules of the operation semantics are silent rules, where the transition is not labeled. Those rules do not introduce non-determinism because their applicability is disjoint from the rules for labelled transitions and from each other.

Next we give a set of operational rules for StAC_i programs.

Activity

We assume that an activity is a relation from states to states, and write $\sigma \xrightarrow{A} \sigma'$ when σ is related to σ' by A. The execution of an activity within an StAC_i process imposes a change in the state, leaving the compensation function unchanged:

$$\frac{\sigma \xrightarrow{A} \sigma'}{(A,\, C,\, \sigma) \;\xrightarrow{A}\; (0,\, C,\, \sigma')}$$

Condition

In this case the execution of a process P is guarded by a boolean function b. If the result of applying b to the current state is $true$, then P is executed:

$$\frac{(P,\, C,\, \sigma) \;\xrightarrow{A}\; (P',\, C',\, \sigma') \quad \wedge \quad b(\sigma) = true}{(b \rightarrow P,\, C,\, \sigma) \;\xrightarrow{A}\; (P',\, C',\, \sigma')}$$

If the result of b is false then skip is executed, and both compensation function and state remain unchanged:

$$\frac{b(\sigma) = false}{(b \rightarrow P,\, C,\, \sigma) \;\rightarrow\; (0,\, C,\, \sigma)}$$

Recursion

In the recursive call of a process N (N is a process identifier), $rec(N)$ is substituted by the term of the left-side of the equation $N = P$:

$$\frac{N = P}{(rec(N),\, C,\, \sigma) \;\rightarrow\; (P,\, C,\, \sigma)}$$

Sequence

This rule states that in a sequence of processes $P; Q$, an order is imposed, so the process P is executed first and only then Q can be executed:

$$\frac{(P,\, C,\, \sigma) \;\xrightarrow{A}\; (P',\, C',\, \sigma')}{(P; Q,\, C,\, \sigma) \;\xrightarrow{A}\; (P'; Q,\, C',\, \sigma')}$$

Executing 0 in sequence with P is the same as executing P alone:

$$\frac{}{(0; P,\, C,\, \sigma) \;\rightarrow\; (P,\, C,\, \sigma)}$$

Parallel
Parallel processes can be executed in an arbitrary order. The parallel process $P \parallel Q$ terminates when both P and Q terminate:

$$\frac{(P,\, C,\, \sigma) \ \xrightarrow{A}\ (P',\, C',\, \sigma')}{(P \parallel Q,\, C,\, \sigma) \ \xrightarrow{A}\ (P' \parallel Q,\, C',\, \sigma')} \qquad \frac{(P,\, C,\, \sigma) \ \xrightarrow{A}\ (P',\, C',\, \sigma')}{(Q \parallel P,\, C,\, \sigma) \ \xrightarrow{A}\ (Q \parallel P',\, C',\, \sigma')}$$

Executing a process P in parallel with 0 is equivalent to just executing P:

$$\frac{}{(P \parallel 0,\, C,\, \sigma) \ \rightarrow\ (P,\, C,\, \sigma)} \qquad \frac{}{(0 \parallel P,\, C,\, \sigma) \ \rightarrow\ (P,\, C,\, \sigma)}$$

Generalised Parallel
The rules for parallel are generalised for set X:

$$\frac{(P_{x_1},\, C,\, \sigma) \ \xrightarrow{A}\ (P'_{x_1},\, C',\, \sigma') \ \wedge \ x_1 \in X}{(\parallel_{\, x \in X}.P_x,\, C,\, \sigma) \ \xrightarrow{A}\ (\parallel_{\, x \in (X - \{x_1\})}.P_x \ \parallel P'_{x_1},\, C',\, \sigma')}$$

Choice
In the process $P[]Q$ only one of the processes P or Q is executed:

$$\frac{(P,\, C,\, \sigma) \ \xrightarrow{A}\ (P',\, C',\, \sigma')}{(P[]Q,\, C,\, \sigma) \ \xrightarrow{A}\ (P',\, C',\, \sigma')} \qquad \frac{(P,\, C,\, \sigma) \ \xrightarrow{A}\ (P',\, C',\, \sigma')}{(Q[]P,\, C,\, \sigma) \ \xrightarrow{A}\ (P',\, C',\, \sigma')}$$

The choice between process 0 and any other process P is equivalent to the single process P:

$$\frac{}{(P[]0,\, C,\, \sigma) \ \rightarrow\ (P,\, C,\, \sigma)} \qquad \frac{}{(0[]P,\, C,\, \sigma) \ \rightarrow\ (P,\, C,\, \sigma)}$$

Generalised Choice
This operator extends choice over a set X:

$$\frac{(P_{x_1},\, C,\, \sigma) \ \xrightarrow{A}\ (P'_{x_1},\, C',\, \sigma') \ \wedge \ x_1 \in X}{([]_{x \in X}.P_x,\, C,\, \sigma) \ \xrightarrow{A}\ (P'_{x_1},\, C',\, \sigma')}$$

Compensation Pair
In the compensation pair $P \div Q$, an evolution in process P does not alter process Q:

$$\frac{(P,\, C,\, \sigma) \ \xrightarrow{A}\ (P',\, C',\, \sigma')}{(P \div_i Q,\, C,\, \sigma) \ \xrightarrow{A}\ (P' \div_i Q,\, C',\, \sigma')}$$

The rule below adds the compensation process Q to the compensation function C, which only happens when the process P finishes:

$$\frac{}{(0 \div_i Q,\, C,\, \sigma) \ \rightarrow\ (0,\, C[i := Q; C(i)],\, \sigma)}$$

The notation $C[i := P]$ denotes that the compensation process of the thread i is set to P. In the above rule the expression $C[i := Q; C(i)]$ means that the compensation thread i is set to Q in sequence with the previous compensation process of i, implying the order of the compensation process is the reverse of the order of execution of the processes.

Compensate
In the next rule, the operator $\boxtimes_i$ causes the compensation process of level i to be executed, and also resets that compensation process to 0:

$$\frac{}{(\boxtimes_i,\ C,\ \sigma)\ \rightarrow\ (C(i),\ C[i := 0],\ \sigma)}$$

Commit
The operator $\boxdot_i$ clears the compensation process of level i to *skip*:

$$\frac{}{(\boxdot_i,\ C,\ \sigma)\ \rightarrow\ (0,\ C[i := 0],\ \sigma)}$$

Merge
The operator $J \rhd i$ merges all compensation processes threads of set J into the compensation process of thread i:

$$\frac{}{(J \rhd i,\ C,\ \sigma)\ \rightarrow\ (0,\ C[i := (||_{j\in J}.C(j)); C(i), J := 0],\ \sigma)}$$

In the above rule the expression $J := 0$ denotes attributing to all elements of set J the process 0, i.e., $\{j := 0 \mid j \in J\}$. J must be disjoint from i.

5.1 Translation from StAC to StAC$_i$

Instead of defining the semantics of the StAC language, we define a translation from StAC to StAC$_i$, so the interpretation of an StAC process P is given in terms of StAC$_i$ by the translation function.

Next we present function $\mathcal{T}$ that translates an StAC process into an StAC$_i$ process,

$$\mathcal{T} : L(\mathrm{StAC}) \times I \rightarrow L(\mathrm{StAC}_i)$$

where $L(\mathrm{SAC})$ and $L(\mathrm{SAC}_i)$ represent StAC and StAC$_i$ languages, and I is a infinite set of indexes. The parameter I is necessary in order to define $\mathcal{T}$ recursively. To translate a process P we have to select an index $i \in I$, and then $\mathcal{T}(P, i)$ will return an StAC$_i$ process.

The next translation rules are fairly simple. For example, to translate a sequential process $P; Q$ with an index i is necessary to translate each process P

and Q with the same index i.

$$
\begin{array}{ll}
\mathcal{T}(A,i) & = A \\
\mathcal{T}(b \rightarrow P,i) & = b \rightarrow \mathcal{T}(P,i) \\
\mathcal{T}(rec(P),i) & = rec(\mathcal{T}(P,i)) \\
\mathcal{T}(P;Q,i) & = \mathcal{T}(P,i);\mathcal{T}(Q,i) \\
\mathcal{T}(P[]Q,i) & = \mathcal{T}(P,i)[]\mathcal{T}(Q,i) \\
\mathcal{T}([]_{x\in X}.P_x,) & = []_{x\in X}.\mathcal{T}(P_x,i) \\
\mathcal{T}(P \div Q,i) & = P \div_i Q \\
\mathcal{T}(\boxtimes,i) & = \boxtimes_i \\
\mathcal{T}(\boxdot,i) & = \boxdot_i \\
\mathcal{T}(0,i) & = 0
\end{array}
$$

The following set of rules are more complex. The main difficulty in the translation is parallel processes and their compensation information. Since we do not know the order of execution of $P \parallel Q$ it implies that we also do not know in which order their compensation should be executed. The solution is to create a new thread for each parallel process, so their compensation process is also a parallel process. In the end both new threads, j and k, are merged into the previous thread i.

$$
\begin{array}{ll}
\mathcal{T}(P \parallel Q,i) & = \boxdot_{\{j,k\}};(\mathcal{T}(P,j) \parallel \mathcal{T}(Q,k));\{j,k\} \rhd i \\
\mathcal{T}(\parallel_{x\in X}.P_x,i) & = \boxdot_J;(\parallel_{x\in X}.\mathcal{T}(P_x,j_x));J \rhd i
\end{array}
$$

where j and k are new indexes, and $J = \{j_x \mid x \in X\}$ is a set of new indexes such that $x \neq x' \Rightarrow j_x \neq j_{x'}$. The notation $\boxdot_J$ is a simplification for $\parallel_{j_x\in J}.\boxdot_{j_x}$. The final merge means that the compensation processes of the parallel processes are retained (unless they have been explicitly committed).

In the last rule we translate the compensation scoping $[P]$. The scoping brackets are translated to a new thread of compensation that in the end is merged in to the previous index:

$$\mathcal{T}([P],i) = \boxdot_j;\mathcal{T}(P,j);\{j\} \rhd i$$

where j is a new index.

To clarify the translation described above, we will exemplify the translation of a simple process,

$$
\begin{array}{rcl}
\mathcal{T}(A \div A' \parallel B \div B');C \div C',i) & = & \mathcal{T}(A \div A' \parallel B \div B',i);\mathcal{T}(C \div C',i) \\
& = & \boxdot_{\{j,k\}};(\mathcal{T}(A \div A',j) \parallel \mathcal{T}(B \div B',k));\{j,k\} \rhd i;C \div_i C' \\
& = & \boxdot_{\{j,k\}};(A \div_j A' \parallel B \div_k B');\{j,k\} \rhd i;C \div_i C'
\end{array}
$$

The first rule applies the function $\mathcal{T}$ to both sequential processes $A \div A' \parallel B \div B'$ and $C \div C'$. To translate the parallel process it is necessary to create a new thread for each parallel process. After the translation of both parallel processes, the new threads are merged into the initial thread.

6 Integrating StAC_i and B

The operational semantics rules at Section 5 allow us to consider a process as a Labelled Transition System (LTS) as in [6]. Besides that, from [2] we have that the B machine that describes the activities can be viewed also as an LTS. For those reasons, the semantics of the integration of StAC_i and B will be defined based on the operational semantics.

A B machine can be viewed as an LTS, where the state space is represented by the cartesian product of the types of state of the machine variables, labels are represented by the operations names and the transitions are represented by the operations.

The semantics of B operations is given in terms of weakest preconditions. For a statement S and postcondition Q, $[S]Q$ represents the weakest precondition under which S is guaranteed to terminate in a state satisfying Q.

In order to define when a transition is allowed by a B operation, we use the notion of conjugate weakest precondition defined as follows:

$$\langle S \rangle Q \mathrel{\hat{=}} \neg [S] \neg Q.$$

$\langle S \rangle Q$ represents the weakest precondition under which it is possible for S to establish Q (as opposed to the guarantee offered by $[S]Q$). If the B machine has an operation of the form

$$A \mathrel{\hat{=}} S,$$

where A is the operation identifier and S is a B AMN statement on the machine state σ, then the transition

$$\sigma \xrightarrow{A} \sigma'$$

is possible provided

$$v = \sigma \quad \Rightarrow \quad \langle S \rangle (v = \sigma').$$

Here v represents the variables of the state machine.

A parameterised B operation of the form

$$A(x) \mathrel{\hat{=}} S$$

represents a set of activity definitions with labels of the form $A.i$, and the activity $A.i$ is given by the statement $x := i; S$.

7 Conclusions

The aim of the collaboration with IBM was to formalise the concepts of the structured ordering of activities and activity compensation. We have achieved that by defining a language (StAC_i) that specifies a system as a set of process equations. The language permits choice, parallel and sequential behaviours. But most important, StAC_i formalises the concept of compensation. Each process can have a compensation process, that is invoked when its actions need to be compensated. There are special processes called activities that represent very simple computations. Instead of extending StAC_i to include variables and expressions we used the B notation to specify activities. The specification of a system has two components, a set of process equations and a B machine describing the activities. The B machine includes a state, operations on the state and boolean operations. The semantics of the StAC_i language and its integration with B is justified through an operational semantics.

The work presented in [3] has similar aims, but it does not have the concept of compensation.

Future work includes the translation of StAC processes into a B machine. Having the overall system specified in B would allow the specification to be animated and appropriate proof obligations to be generated (which is already possible for the activities). An experimental translation has been devised, but it is necessary to provide a formal proof that the translated specification is equivalent to the combined StAC_i and B specification.

Another important extension to the present work is to support the refinement of the B machine, that specifies the overall system, to compositions of EJBs. This would guarantee the correctness of all of the development steps, from the abstract specification (StAC_i and B) to the implementation (Java code).

Acknowledgments

We would like to thank Peter Henderson, Iain Houston, Mandy Chessell, David Vines and Catherine Griffin for their contribution.

References

1. J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. M.J. Butler. csp2B: A practical approach to combining CSP and B. Technical Report DSSE-TR-99-2, Dept. of Electronics and Computer Science, Univ. of Southampton, 1999.
3. H. Eertink, W. Janssen, P.O. Luttighuis, W. Teeuw, and C. Vissers. A business process design language. In *FM'99 – Formal Methods*, pages 76–95. Springer-Verlag, 1999.
4. C.A.R Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
5. K. Lano. *The B Language and Method: A Guide to Practical Formal Development.* Formal Approaches to Computing and Information Technology. Springer-Verlag, 1996.
6. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
7. R. Monson-Haefel. *Enterprise Java Beans.* O'Reilly, 1999.