Using B Refinement to Analyse Compensating Business Processes

Carla Ferreira and Michael Butler

Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton SO17 1BJ, United Kingdom
{cf,mjb}@ecs.soton.ac.uk

Abstract. This paper explores the refinement of compensating business processes, which are modelled in a heterogeneous notation that combines StAC and B. In our refinement approach, the StAC behavioural and compensation information are explicitly embedded in a B machine. As the resulting machine is standard B one can use the B notion of refinement to prove the refinement of business processes. We also show how the Ateleir-B prover can help in constructing the gluing invariant.

1 Introduction

StAC (Structured Activity Compensation) [8] is a formal business process modelling language. The distinctive feature of the language is the concept of compensation, which can be defined as the action taken to correct any errors or when there is a change of plan. The motivation for developing StAC came from a collaboration with IBM concerning the extension of existing notions of compensation for business transactions within the BPBeans enterprise technology [9].

A model of a business process is specified as a set of StAC processes plus a B machine [2]. The StAC processes describe the execution order of the operations and the compensation information, while the B machine describes the state of the system and its basic operations. A model involving compensation does not always give a clear view of the overall properties of the system. Consequently, we need to provide a way of verifying that such a model (involving compensation) satisfies certain properties. We propose to use refinement as a way of verifying StAC specifications, by representing the property as a more abstract system and show that it is refined by the model with compensation.

In this paper we explore a refinement approach where both specification parts – StAC and B – are refined simultaneously. The refinement strategy presented here follows the csp2b approach [5] and combines the StAC processes and the B machine into a new B machine that deals explicitly with compensation and operation ordering. The fact that the resulting machine is standard B allows the use of the B notion of system refinement to validate refinement between StAC specifications. The contribution of this paper is:
- demonstration of refinement in context of heterogeneous specifications, and
- showing how the Atelier-B prover can help in the construction of the gluing invariant.

Also, to simplify the construction of invariant of our case study (the e-bookstore) we started by constructing the invariant for a single client and then generalised it to any number of number clients. This strategy made the determination of the invariant simpler, as Atelier-B suggested most of the invariant clauses.

The next section describes briefly the StAC language. Section 3 describes our strategy to system refinement. Section 4 presents the abstract and concrete specifications of the e-bookstore. In Section 5 we apply our refinement approach to a single client, and Section 6 shows how to extend the results of a single client to a bookstore with any number of clients.

2 StAC Language

The StAC language allows sequential and parallel composition of processes, and the usual process combinators. It also has specific combinators to deal with compensation. Most of the language operators are presented in Table 2. A complete description of StAC, including a formal semantics, can be found in [8, 10].

\[
\text{Process ::= } A \quad \text{(activity label)}
\]

| \(null\) | (null) |
| \(b \rightarrow P\) | (condition) |
| \(\text{rec}(N)\) | (recursion) |
| \(P; Q\) | (sequence) |
| \(P \parallel Q\) | (parallel) |
| \(\parallel x.P_x\) | (generalised parallel) |
| \(P||Q\) | (choice) |
| \(\parallel x.P_x\) | (generalised choice) |
| \(P \div Q\) | (compensation pair) |
| \(\overrightarrow{\text{reverse}}\) | |
| \(\overleftarrow{\text{accept}}\) | |
| \(J \triangleright i\) | (merge) |

Table 1. StAC Syntax

Each activity label \(A\) (in StAC) has an associated activity \(\overset{A}{\rightarrow}\) (in B) representing an atomic change in the state. The sequential construct is a binary operator that composes two processes, \(P\) and \(Q\). In the process \(P; Q\), \(P\) is executed first. When \(P\) completes, \(Q\) is executed. There are two forms of concurrent construct, the binary form, \(P \parallel Q\), which composes two process in parallel, and the generalized form, \(\parallel x.P_x\), which models concurrent invocation of multiple instances of a
process. The choice \( P \parallel Q \) selects whichever of \( P \) or \( Q \) is enabled. Generalised choice extends choice over a set of processes.

A compensation pair \( P \div i Q \) is a grouping of two tasks, where \( P \) is the primary task, and \( Q \) is the compensation task. When a compensation pair is executed, it only executes the primary task. Once the primary task has completed, the compensation task is remembered on compensation task \( i \). We have to differentiate the compensation tasks as a process may have several simultaneous compensation tasks associated with it. A process decides which task to attach compensation activities to, and individual tasks can be accepted or reversed. The instruction that performs compensation on task \( i \) is the reversal operator, \( \boxtimes_i \). For example, consider the following process, where \( A \) and \( B \) are activities:

\[(A \div i B); \boxtimes_i\]

This process will perform activity \( A \) and remember the compensation \( B \) on task \( i \). The reversal instruction will then cause compensation activity \( B \) to be executed. A sequence of compensation pairs is compensated in reverse order, so the process:

\[(A_1 \div i B_1); (A_2 \div i B_2); (A_3 \div i B_3); \boxtimes_i\]

executes \( A_1, A_2 \) and \( A_3 \) sequentially and then, because of the reversal on task \( i \), executes \( B_3, B_2, B_1 \) sequentially. The acceptance operator, \( \boxtimes_i \), indicates that currently remembered compensations of task \( i \) should be forgotten as they will no longer be required. For example the process:

\[(A_1 \div i B_1); \boxtimes_i; (A_2 \div i B_2); \boxtimes_i\]

performs \( A_1 \) followed by \( A_2 \) and then performs the compensation \( B_2 \). Compensation \( B_1 \) is not performed as it will have been removed by the acceptance instruction before the reversal. Until now we have only used a single compensation task, but the next example illustrates the use of several compensation tasks:

\[(A_1 \div i B_1); (A_2 \div j B_2); \boxtimes_i; (A_3 \div j B_3); \boxtimes_j\]

This process will invoke \( A_1, A_2 \) and then the reversal causes compensation \( B_1 \) to be invoked. Compensation \( B_2 \) will not be invoked at this stage as it is on compensation task \( j \) and only compensation task \( i \) is invoked by the first reversal operator. After the first compensation, activity \( A_3 \) is performed. Reversal is then invoked on compensation task \( j \) which causes \( B_3 \) followed by \( B_2 \) to be executed.

An important operator in \( \text{StAC} \) is the merge operator. The expression \( J \triangleright i \), where \( J \) is set of indices, merges all compensation tasks belonging to \( J \) into the compensation task \( i \). When merging compensation tasks, those tasks are merged in parallel. For example the process

\[(A_1 \div i B_1); (A_2 \div j B_2); (A_3 \div k B_3); \{i, j \} \triangleright k\]

\[\text{Informally we can say that the compensation information is maintained as a function that for each task index returns the associated compensation process.}\]
initially it executes $A_1$, $A_2$, and $A_3$ and then merges compensation tasks $i$ and $j$ into compensation task $k$. Joining compensation tasks $i$ and $j$ results in the parallel process $B_1 \parallel B_2$, which will be put in front of the compensation task $k$, giving $(B_1 \parallel B_2); B_3$ as the resulting compensation for task $k$.

2.1 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 s/he can either pay or abandon the bookstore, in the later case all selected books have to be returned to the shelf.

The e-bookstore is defined as an infinite set of parallel *Client* processes, over the set $CLIENT$ of all possible on-line clients:

$$Bookstore = \| c \in CLIENT . Client(c)$$

The first activity in process *Client* is *Arrive*. 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 reversal is invoked causing the return of all books in the client’s basket to the shelf. Notice that each client has an independent compensation task $c$, so the reversal $\mathbb{E}_c$ will only return the books of client $c$. If the client decides to pay for his/her order, *Pay* will process the client’s card. If the card is rejected, the reversal instruction will be invoked. *Exit* represents the packaging of all books in the client’s basket.

$$Client(c) = Arrive(c);$$
$$ChooseBooks(c);$$
$$\{$$
$$\{$$
$$Quit(c); \mathbb{E}_c$$
$$\}$$
$$\}$$
$$Pay(c); (\neg accepted(c) \rightarrow \mathbb{E}_c);$$
$$Exit(c)$$

To select books the client iterates over the selection of individual books until *Checkout* is invoked:

$$ChooseBooks(c) = Checkout(c) \parallel (ChooseBook(c); ChooseBooks(c))$$

In *ChooseBook* a new compensation process is created for each book selected. The new compensation $c_1$ is only related to the selected book. Within *ChooseBook* there is a compensation pair, *AddBook* compensated by *ReturnBook*, and the compensation process is only executed if adding that book to the basket exceeds the budget. In this case executing the compensation task implies returning the book that has just been added to the basket, rather than all books in the basket. If the budget is not exceeded, the compensation is preserved by the merge constructor.

$$ChooseBook(c) = \\{ b \in book . (AddBook(c, b) \div c_1 ReturnBook(c, b));$$
$$\text{overBudget}(c) \rightarrow \mathbb{E}_{c_1}; c_1 \triangleright c$$
3 StAC Refinement

The strategy we have defined for refinement of StAC specifications is described in Fig. 1, and it is based on the esp2b [5] approach. The first step (a) extracts a State Transition Diagram (STS) from a set of StAC processes. The STS describes the execution order of the activities. In the second step (b) the information of the STS is explicitly included in the original B specification. The resulting $M_B$ and $N_B$ specifications are standard B machines that deal explicitly with compensation and operation ordering. With this approach, to prove that $N$ is a refinement of $M$, it is necessary to build both $M_B$ and $N_B$ machines and prove within the B method that $N_B$ is a refinement of $M_B$. Because the resulting B machines are standard B we have used Atelier-B to generate the proof obligations and its prover to assist in proving those obligations.

In this paper we apply the approach of Fig. 1 to the e-bookstore example, and explain by example the embedding of StAC into B (for a detailed description see [10]).

4 Case Study

The e-bookstore example will be used to show how refinement can be use to verify system properties, and to study the applicability of the proposed refinement strategy. We will start by defining a more abstract specification of the e-bookstore that provides a simplified functionality of the system without using compensation. This specification captures the basic properties that must be preserved by the system. Some of the e-bookstore properties are: a client cannot exceed his/her predefined budget; books are transferred from the shelf to the basket; transactions can be accepted or rejected; if rejected, books are returned to the shelf. Notice that the concrete e-bookstore (Section 2.1) does not always preserve this property. When adding a book to the basket, the system only verifies if the budget was exceeded after the book is already in the basket. If the budget was exceeded, compensation will be invoked restoring the system consistency.
The abstract e-bookstore will be called Bookstore0, while the concrete specification will be renamed Bookstore1. Ultimately we want to prove that

\[ \text{Bookstore0} \sqsubseteq \text{Bookstore1} \]

by using the event-B [i] notion of system refinement. In event-B the concrete system may introduce extra operations that refine skip.

4.1 Abstract model

The abstract e-bookstore is defined as an infinite set of parallel Client0 processes:

\[ \text{Bookstore0} = \bigparallel c \in \text{CLIENT} . \text{Client0}(c) \]

Process Client0 is a sequential process, which starts with activity Arrive that initialises the client information. The next activity is Checkout, which represents a client choosing simultaneously all the books s/he wants to buy. Activity Checkout is followed by a choice between paying for the books or abandoning the bookstore without buying any books. The Pay activity verifies whether the card of the client is accepted and if the card is rejected the books in the basket will be returned. In the Quit activity the client’s basket and its content will be returned to the shelves. The last process Exit represents the packaging of the all books in the client’s basket.

\[ \text{Client0}(c) = \text{Arrive}(c); \text{Checkout}(c); (\text{Pay}(c) \parallel \text{Quit}(c)); \text{Exit}(c) \]

The state of the Bookstore0 machine has two sets: CLIENT that represents all clients that can be on-line simultaneously; and BOOK that represents all books available in the bookstore. Variables basket, budget, and accepted are partial functions that return for each client, respectively, the selected books, the allowed spending money, and the card status. These functions have the same domain, which represents the set of clients accessing on-line the bookstore. The variable shelf returns for each book its availability, and price contains the price of each book. The first clause in the invariant states that if a client is on-line, then s/he must have a basket, a budget, and a card status. The second clause states that every on-line client must keep his/her basket within the predefined budget.

MACHINE Bookstore0
SETS CLIENT, BOOK
VARIABLES basket, budget, accepted, shelf, price
DEFINITIONS
\[ \text{overBudget}(c) \equiv \sum b . (b \in \text{basket}(c) \mid \text{price}(b)) \geq \text{budget}(c) \]
\[ \text{inBudget}(s, c) \equiv \sum b . (b \in s \mid \text{price}(b)) \geq \text{budget}(c) \]

INVARIANT
\[ \text{basket} \in \text{CLIENT} \rightarrow \mathcal{F} (\text{BOOK}) \land \]
\[ \text{budget} \in \text{CLIENT} \rightarrow \mathbb{N}_0 \land \]
\[ \text{accepted} \in \text{CLIENT} \rightarrow \text{BOOL} \land \]
\[ \text{shelf} \in \text{BOOK} \rightarrow \mathbb{N} \land \]
\[ \text{price} \in \text{BOOK} \rightarrow \mathbb{N}_0 \land \]
\[ \text{dom(basket)} = \text{dom(budget)} = \text{dom(accepted)} \land \]
\[ \forall c \in \text{CLIENT} . c \in \text{dom(basket)} \Rightarrow \neg \text{overBudget}(c) \]
Next, we will describe in detail most of the Bookstore operations. If client \( c \) is not already on-line, \textbf{Arrive} will initialise the new client’s information.

\begin{verbatim}
Arrive(c : CLIENT) \^ \nSELECT c \notin \text{dom}(basket) \THEN \n\text{ANY a WHERE a \in \mathbb{N} \THEN \n\hspace{1cm} \text{basket} := \text{basket} \cup \{c \mapsto \emptyset\} \parallel \n\hspace{1cm} \text{budget} := \text{budget} \cup \{c \mapsto a\} \parallel \n\hspace{1cm} \text{accepted} := \text{accepted} \cup \{c \mapsto \text{FALSE}\} \n\hspace{1cm} \text{END} \n\hspace{1cm} \text{END} \n\end{verbatim}

\textbf{Checkout} is enabled for clients that are already on-line, and it chooses non-deterministically a set of books that are within the client’s budget and in stock, and puts them in the basket. This operation gives a very simplified view of choosing books in a bookstore, usually a client would want to choose the books him/herself.

\begin{verbatim}
Checkout(c : CLIENT) \^ \nSELECT c \in \text{dom}(basket) \THEN \n\text{ANY books WHERE books \subseteq BOOK \land \text{inStock}(books) \land \text{inBudget}(books, c) \THEN \n\hspace{1cm} \text{basket}(c) := \text{books} \parallel \n\hspace{1cm} \text{shelf} := \text{shelf} \leftarrow \lambda(book).(book \in books \mid \text{shelf}(book) - 1) \n\hspace{1cm} \text{END} \n\hspace{1cm} \text{END} \n\end{verbatim}

Operation \textbf{Pay} describes the payment of the books at a very abstract level. \textbf{Pay} performs two actions, verifying the client’s card and returning the books in the basket to the shelves, if the card is rejected.

\begin{verbatim}
Pay(c : CLIENT) \^ \nSELECT c \in \text{dom}(basket) \THEN \n\text{CHOICE} \n\hspace{1cm} \text{accepted}(c) := \text{TRUE} \n\text{OR} \n\hspace{1cm} \text{accepted}(c) := \text{FALSE} \parallel \n\hspace{1cm} \text{basket}(c) := \emptyset \parallel \n\hspace{1cm} \text{shelf} := \text{shelf} \leftarrow \lambda(book).(book \in \text{basket}(c) \mid \text{shelf}(book) + 1) \n\hspace{1cm} \text{END} \n\hspace{1cm} \text{END} \n\end{verbatim}

\textbf{Quit} represents the client leaving the bookstore without buying any books, so it just returns the books in the client’s basket to the shelf. The last operation \textbf{Exit} does not alter any state variable it just assigns the basket to an output variable.

\footnote{The notation \( A(x : X) \equiv S \) is an abbreviation for \( A(x) \equiv \text{PRE} x : X \THEN S \END \).}
4.2 Concrete Model

The StAC specification of the concrete e-bookstore was already presented in Section 2.1, so we will continue by describing the Bookstore1 machine. The main difference between the abstract and concrete specifications is that in the later books are added one at the time, and compensation is used to return books. The state of Bookstore1 is similar to the abstract state (each abstract variable \( v \) will be replaced by a concrete variable \( v1 \)), so we will only describe the concrete activities that are not identical to their abstract representations. AddBook is enabled if \( c \) is a on-line client, \( b \) is not in the client’s basket and it is available. If all conditions are met, \( b \) is added to the basket of client \( c \). The operation ReturnBook has similar enabling conditions, but instead it removes a book from the client’s basket.

\[
\text{AddBook}(c: \text{CLIENT}, b: \text{BOOK}) \triangleq \\
\text{SELECT } c \in \text{dom}(\text{basket}1) \land b \not\in \text{basket}1(c) \land \text{shelf}1(b) > 0 \text{ THEN} \\
\text{basket}1(c) := \text{basket}1(c) \cup \{b\} \land \\
\text{shelf}1(b) := \text{shelf}1(b) - 1 \\
\text{END}
\]

Checkout is used in process ChooseBooks to exit its recursive definition, so it does not need to perform any explicit action. Operation Quit is similar to operation Checkout. Quit is used to determine which action the client wants to perform, quit the bookstore or pay the books.

\[
\text{Checkout}(c: \text{CLIENT}) \triangleq \text{SELECT } c \in \text{dom}(\text{basket}1) \text{ THEN } \text{skip} \text{ END}
\]

Both accepted1 and overBudget are used as guards of conditional processes, so they are specified in B as boolean expressions (process guards do not change the machine state). Operation Pay sets the variable accepted1, which is used to trigger the execution of the compensation process when the card is rejected.

\[
\text{Pay}(c: \text{CLIENT}) \triangleq \\
\text{SELECT } c \in \text{dom}(\text{basket}1) \text{ THEN} \\
\text{CHOICE } \text{accepted}1(c) := \text{TRUE} \text{ OR } \text{accepted}1(c) := \text{FALSE} \text{ END} \\
\text{END}
\]

Operation Pay is described as a choice between attributing the value TRUE or FALSE to accepted1, depending on the card being accepted or rejected.

5 Dealing with Single Clients

Both abstract and concrete e-bookstore are generalised parallel processes, executing concurrently all clients accessing on-line the bookstore. Therefore, to simplify the determination of the gluing invariant, we decided to deal first with a single client and later extend the invariant for any number of concurrent clients.
5.1 Constructing the Client0B machine

To prove that Client1 refines Client0 we will follow the steps described in Fig. 1. Fig. 2 shows the STS extracted from process Client0. A client must first execute operation Arrive, followed by Checkout. Next, there are two alternative transitions labelled Pay and Quit. These transitions are the STS representation for the choice between the activities Pay and Quit. Last, a client has to execute operation Exit.

Next, we need to extend the Client0 machine to deal the behavioural information of its STS. The extended machine Client0B has two additional components: a set STATE that contains the states of the STS; and a variable state that will keep track of the machine current state.

MACHINE Client0B

SETS
BOOK;
STATE = \{a1, a2, a3, a4, a5\}

VARIABLES basket, budget, accepted, shelf, price, state

INARIANT
\[ \vdots \land state \in \text{STATE} \land \]
\[ \text{budget} \geq \sum_{(\text{book})}(\text{book} \in \text{basket} \mid \text{price}(\text{book})) \]
The last clause in the invariant guarantees the initial requirement of the client buying within the budget.

Because process \textit{Client}0 does not deal with compensation, we only have to extend each operation with a \texttt{SELECT} statement that assures the operation will be executed in the order defined by the STS of Fig. 2. In \texttt{Checkout} the \texttt{SELECT} statement enables the operation when the system is on state \texttt{a2}. The remaining operations are extended in a similar way.

\texttt{Checkout} \seteq \begin{align*}
\texttt{SELECT} \quad \texttt{state} = \texttt{a2} \quad \texttt{THEN} \quad \texttt{state} := \texttt{a3} \quad \texttt{END} \quad || \\
\text{\texttt{ANY}} \quad \texttt{books} \quad \text{\texttt{WHERE}} \quad \texttt{books} \subseteq \texttt{BOOK} \land \texttt{inStock} (\texttt{books}) \land \neg \texttt{overBudget} (\texttt{books}) \\
\texttt{THEN} \\
\quad \texttt{basket} := \texttt{books} \quad || \\
\quad \texttt{shelf} := \texttt{shelf} \triangleq \lambda (\texttt{book}).(\texttt{book} \in \texttt{books} \mid \texttt{shelf} (\texttt{book}) - 1) \\
\texttt{END}
\end{align*}

5.2 \textbf{Constructing the Client1\textsubscript{B} machine}

The STS extracted from the process \textit{Client1} is presented in Fig. 3. This STS deals with compensation operators, like compensation pair, merge, and reversal.
The operators merge and compensation pair are represented by a single transition labelled with the operator name, as for example the transition from $c_2$ to $c_3$ labelled $\text{AddBook}(b) \div_1 \text{ReturnBook}(b)$ (we have changed the compensation task identifier $c_1$ and $c$ to $i_1$ and $i$ to avoid confusion with the STS concrete states).

The reversal has a more complex representation, because it has to invoke sequentially the activities in the compensation. In the general form, $\mathbb{R}_j$ is represented by the following STS:

$$\begin{align*}
S1 & \xrightarrow{[\text{size}(C(j)) > 1]} S2 \xrightarrow{[\text{size}(C(j)) = 0]} S3 \\
S1 & \xrightarrow{[\text{size}(C(j)) = 1]} S2 \xrightarrow{\text{first}(C(j))} S3
\end{align*}$$

where $C \in \text{INDEX} \rightarrow \text{seq}(\text{ACTIVITY})$ is the compensation function that for each task index returns a sequence of compensation activities. The occurrence of $\mathbb{R}_j$ causes the state to evolve from $S1$ to $S2$. In state $S2$ only one of the three transitions will be enabled, depending on the number of compensation activities on task $j$. If compensation task $j$ is empty, the STS will evolve to $S3$ without performing any action. If the compensation task $j$ has a single compensation activity, that activity will be executed and the STS will evolve to $S3$. Otherwise, the compensation activities will be executed sequentially until a single activity remains. A more systematic approach to the construction of STS from St-AC processes is presented in [10].

In the STS of Fig. 3 we can simplify the transitions that occur after the reversal (starting from $c_4$ and $c_8$). Those transitions can invoke directly the operation $\text{ReturnBook}$, because it is the only operation name in both compensation tasks:

The transitions that occur after state $c_4$ can be simplified even further, because we know that the sequence that represents task $i_1$ has exactly one element, and that this element is $\text{ReturnBook}$. Therefore state $c_4$ has a single transition that causes the last book added to the basket to be returned to the shelves. So, all guards and dotted transitions can be ignored.

Now that we have the STS for the $\text{Client1}$ process, the next step is to embed it into the $\text{Client1}$ machine. Two new sets and variables where added to the state of $\text{Client1}$ machine. $\text{STATE1}$ has the states used in the STS and $\text{INDEX}$ has the compensation task indices used in the o-bookstore processes.
Given that \texttt{ReturnBook} is the only compensation operation in the system we only need to “store” in compensation function \( C \) the value of the argument of \texttt{ReturnBook}. When the primary activity of the compensation pair \texttt{AddBook}(b) \vdash \texttt{ReturnBook}(b) \) occurs, it is necessary to keep the value of \( b \) stored so that if a reversal occurs \texttt{ReturnBook} \ will be invoked with the correct argument.

\begin{verbatim}
REFINEMENT Client1_B
REFINES Client0_B
SETS
\text{STATE1} = \{ c1, c2, \ldots, c10, c11 \};
\text{INDEX} = \{ i, \text{i1} \} 
VARIABLES basket1, budget1, accepted1, shelf1, price1, state1, C
INVARINT
\hspace{1cm} \ldots \wedge
\hspace{1cm} \text{state1} \in \text{STATE1} \wedge
\hspace{1cm} C \in \text{INDEX} \rightarrow \text{seq(BOOK)}
\end{verbatim}

The concrete operation \texttt{Checkout} is used to exit the recursive process of adding single books to the basket, and it becomes enabled in state \( c2 \) and its execution causes the state to evolve to \( c6 \).

\texttt{Checkout} \hspace{1cm} \begin{verbatim}
\begin{array}{l}
\text{SELECT state1 = c2 \ THEN state1 := c6 END}
\end{array}
\end{verbatim}

The operation \texttt{Quit} also does not perform any action besides changing the machine state.

The extensions to operation \texttt{AddBook} are more complex because \texttt{AddBook} is the primary task of \texttt{AddBook}(b) \vdash \texttt{ReturnBook}(b). Therefore, the parameter \( b \) has to be added to compensation task \text{i1} whenever \texttt{AddBook} is executed.

\texttt{AddBook}(b : BOOK) \hspace{1cm} \begin{verbatim}
\begin{array}{l}
\text{SELECT state1 = c2 \ THEN state1 := c3 \ \parallel C(i1) := b \leftarrow C(i1)}
\end{array}
\end{verbatim}

\begin{verbatim}
\parallel
\begin{array}{l}
\text{SELECT } b \notin \text{basket1} \ \wedge \ \text{shelf1}(b) > 0 \ \text{THEN}
\hspace{1cm} \text{basket1} := \text{basket1} \cup \{ b \} \ \parallel
\hspace{1cm} \text{shelf1}(b) := \text{shelf1}(b) - 1
\end{array}
\end{verbatim}

END

Operation \texttt{ReturnBook} is a compensation action, so it will be invoked after the occurrence of the reversal in states \( c4 \) and \( c8 \). In state \( c4 \) the operation is called after the reversal of task \text{i1} and it will enabled if the parameter \( b \) is equal to the book on top of \( C(i1) \). In state \( c8 \) the operation \texttt{ReturnBook} is successively

\footnote{\texttt{C(i1)} does not need to be a sequence as it contains at most one element.}
invoked until the compensation task i is empty.

\[
\text{ReturnBook}(b : \text{BOOK}) = \\
\begin{align*}
\text{SELECT state1} & = c4 & \text{size(C(i1))} = 1 & \text{first(C(i1))} = b & \text{THEN} \\
C(i1) & := \text{tail}(C(i1)) \parallel \\
\text{state1} & := c5 \\
\text{WHERE state1} & = c8 & \text{size(C(i))} \geq 1 & \text{first(C(i))} = b & \text{THEN} \\
C(i) & := \text{tail}(C(i)) \parallel \\
\text{IF size(C(i))} & = 1 & \text{THEN state1} & := c10 & \text{END} \\
\end{align*}
\]

The \text{Reverse}, \text{Merge} and \text{Null} are new operations to be added to \text{Client1B} machine. The \text{Reverse} may be invoked in three different states, c3, c7, and c9. In each of one this states, and if the its other conditions hold, the reversal will cause the state to evolve.

\[
\text{Reverse}(\text{index} : \text{INDEX}) = \\
\begin{align*}
\text{SELECT state1} & = c3 & \text{index} = i1 & \text{overBudget(basket1)} & \text{THEN state1} := c4 \\
\text{WHEN state1} & = c7 & \text{index} = i & \text{THEN state1} := c8 \\
\text{WHEN state1} & = c9 & \text{index} = i & \text{~accepted} & \text{THEN state1} := c8 \\
\end{align*}
\]

The \text{Merge} is enabled on state c5, and if applied with the expected task indices. \text{Merge} will put compensation task i1 on top of task i and clear task i1.

\[
\text{Merge}(\text{index}1 : \text{INDEX}, \text{index}2 : \text{INDEX}) = \\
\begin{align*}
\text{SELECT index}1 & = i1 & \text{index}2 = i & \text{state1} = c5 \\
\text{THEN} \\
C & := \{\text{index}2 \mapsto C(\text{index}1)^{-1}C(\text{index}2), \text{index}1 \mapsto []\} \parallel \\
\text{state1} & := c2 \\
\end{align*}
\]

The \text{Null} operator is used when the STS has unlabelled guarded transitions. The STS for \text{Client1} has two empty transitions. In state c3 the empty transition occurs when the book added to basket keeps the basket within the budget, no action has to be done and the client may continue choosing books. In state c9 the client has decided to pay for the books and his/her card was accepted, again no further action needs to be done and the state evolves to c10.

\[
\text{Null} = \\
\begin{align*}
\text{SELECT state1} & = c3 & \text{~overBudget(basket1)} & \text{THEN state1} := c5 \\
\text{WHEN state1} & = c9 & \text{accepted} & \text{THEN state1} := c10 \\
\end{align*}
\]

5.3 Devising an Abstraction Invariant

We need to devise an invariant I that relates the variables of the abstract system to those of the refined system:
Atelier-B was used to generate the proofs obligations and to help construct most of the invariant clauses in an incremental way. We will explain next how we used Atelier-B to help us deducing the invariant.

When applying the Atelier-B automatic prover to a refinement there are two possible outcomes, all proofs are proved (the specification is proven correct) or there are some proof obligations left unproved. When the automatic prover fails to prove an obligation, the user has to examine each failed proof obligation and determine the reason for that failure:

1. The proof obligation is too complex to be done automatically.
2. The proof obligation is impossible to prove with the present invariant clauses.
3. The proof obligation is false, so the refinement claim is invalid.

In the first case, the user has to assist the automatic prover in its demonstration, by using a set of interactive commands provided by Atelier-B. In the second case, the invariant is too weak as its clauses are not sufficient to prove all proof obligations. This can be solved by strengthening the invariant with new clauses. With some specifications, the clauses to be added can be extracted almost directly from unproved obligations. This is what we called earlier “Atelier-B helping to construct the invariant” which is done by strengthening the invariant with the failed proof obligations. In the last case, either the specification or the refinement (or both) have to change.

Initially we just added the clause $C_1$ to the invariant, stating that the concrete
variables \textit{price1}, \textit{budget1} and \textit{accepted1} are equal to the correspondent abstract variables\footnote{This is generated automatically by \textit{Atelier-B} if abstract and concrete variables have the same name.}. This incremental approach of building the invariant is based on \[7\].

\[ C_1 \quad \text{price1} = \text{price} \land \text{budget1} = \text{budget} \land \text{accepted1} = \text{accepted} \]

After using the automatic prover on \textit{Client1B} with the clause \( C_1 \) several proof obligations where left unproved. Fig. 4 shows \textit{Atelier-B} interactive prover applied to one of those unproved obligations, where the user is asked to help the automatic prover discharging the proof obligation \textit{Arrive1}. This proof obligation corresponds directly to \( C_{2.1} \). The other unproved obligations corresponded to the remaining clauses of \( C_2 \). The clauses \( C_2 \) relate the abstract variable \textit{state} to the concrete variable \textit{state1}. Those clauses could be deduced directly from analysing both the abstract and concrete STS. For example, in any of the states \( \{c2, c3, c4, c5\} \) the transition \textit{Arrive} has occurred and the only external operation that may occur next is \textbf{Checkout}, which are the incoming and outgoing transitions of \( a2 \).

\begin{align*}
C_{2.1} & \quad \text{state1} = c1 \Rightarrow \text{state} = a1 \\
C_{2.2} & \quad \text{state1} \in \{c2, c3, c4, c5\} \Rightarrow \text{state} = a2 \\
C_{2.3} & \quad \text{state1} = c6 \Rightarrow \text{state} = a3 \\
C_{2.4} & \quad \text{state1} \in \{c7, c8, c9, c10\} \Rightarrow \text{state} = a4 \\
C_{2.5} & \quad \text{state1} = c11 \Rightarrow \text{state} = a5
\end{align*}

The invariant \( C_3 \) describes in which states both abstract and concrete baskets have the same books. Clause \( C_{3.1} \) was not suggested directly by \textit{Atelier-B}, but it was introduced in the process of interactively proving the proof obligation for \textbf{Checkout}. \( C_{3.1} \) shows that after \textbf{Checkout} both systems baskets are equal, because in the concrete system the client has finished choosing individually each book of \textit{basket1} and in the abstract system a set of books was placed in \textit{basket}. Also in the final state both baskets must have the same books. Clause \( C_{3.2} \) was constructed by \textit{Atelier-B} and it shows that if the client’s card was accepted, both baskets must have the same books.

\begin{align*}
C_{3.1} & \quad \text{state1} \in \{c6, c10\} \Rightarrow \text{basket1} = \text{basket} \\
C_{3.2} & \quad \text{state1} = c9 \land \text{accepted1} = \text{TRUE} \Rightarrow \text{basket1} = \text{basket}
\end{align*}

Clause \( C_{4.1} \) was added in order to preserve clause \( C_{3.1} \) and it says that the abstract basket will be empty after the occurrence of \textbf{Quit}, while the concrete basket still has all the books chosen by the client. The clause \( C_{4.2} \) was constructed directly by \textit{Atelier-B} and it states that after \textbf{Pay} and if the client’s card was not accepted the abstract basket will be empty, because operation \textbf{Pay} removes the books from the basket at the same times the card is rejected. The concrete system operation \textbf{Pay} just verifies the card maintaining all the books in the basket, as they will be removed by invoking the reversal.

\begin{align*}
C_{4.1} & \quad \text{state1} \in \{c7, c8\} \Rightarrow \text{basket} = \emptyset \\
C_{4.2} & \quad \text{state1} = c9 \land \text{accepted1} = \text{FALSE} \Rightarrow \text{basket} = \emptyset
\end{align*}
The clauses C₅ relates the compensation tasks to the concrete basket. Clauses C₅₁ and C₅₂ say that each book on the basket must be in one of the compensation tasks but not in both. The last clause is necessary to prove the two previous clauses.

C₅₁ \( \text{ran}(C(i1)) \cup \text{ran}(C(i)) = \text{basket1} \)

C₅₂ \( \text{ran}(C(i1)) \cap \text{ran}(C(i)) = \emptyset \)

C₅₃ \( \text{state1} \in \{c2, c6, c7, c8, c9\} \Rightarrow C(i1) = [\] \)

Although in the abstract system the value of the books in the basket is always within the budget, in the concrete system after the operation \texttt{AddBook} the basket may exceed the budget, and as a result the last book added to basket must be returned. The fact that there are states in the concrete machine where the budget is exceeded does not breach the abstract invariant, because this only happens with internal operations that are not visible to the abstract machine. Clause C₆₁ asserts that before adding a new book to the basket (state e2) and after returning the last book added if the budget was exceeded (state e5), the basket is within the budget. Clauses C₆₂ and C₆₃ where constructed directly by \texttt{Atelier-B} after we added clause C₆₁. This two clauses show that if the budget was exceeded after the occurrence of \texttt{AddBook}, this was caused by adding the last book to the basket.

C₆₁ \( \text{state1} \in \{c2, e5\} \Rightarrow \text{inBudget(basket1)} \)

C₆₂ \( \text{state1} = c3 \land \text{overBudget(basket1)} \Rightarrow \text{inBudget(basket1) - \{first(C(i1))\}} \)

C₆₃ \( \text{state1} = c4 \Rightarrow \text{inBudget(basket1) - \{first(C(i1))\}} \)

The last clause of the invariant (C₇) says that each book in the abstract and concrete system are either in basket or in the shelf, although the abstract and concrete values may not agree. For example, a book might be in the shelf in the abstract system and in the basket in the concrete system.

C₇ \( \forall \text{book}. \text{book} \in \text{BOOK} \Rightarrow \text{shelf(book) + inBasket(book, basket) = shelf1(book) + inBasket(book, basket1)} \)

### 5.4 Proving the Refinement

We have proved using \texttt{Atelier-B} that the clause C₁ \( \land C₂ \land \cdots \land C₇ \) is a gluing invariant for the refinement of Client₁B by Client₁B. The total number of proof obligations adds up to 202, of those 171 where automatically proved by the prover of \texttt{Atelier-B}. From the remaining 31 proofs, 13 of those were fairly easy to prove by interaction with the prover. For the remaining 18 unproved obligations it was necessary to define a user rule file to assist the automatic prover. The user rules are necessary when the prover rule database does not have rules to deal with a specific type of proof. Most of our rules where concerned with sequences or lambda expressions, for which the rule database had a very limited set of rules. Even with the user rules 4 proofs where difficult and time consuming.

Proving the refinement for a single client was very useful, because it allowed us to develop a gluing invariant in a incremental way. The \texttt{Atelier-B} prover
constructed most of the invariant clauses and by attempting to prove proof obligations for weak gluing invariants we have constructed some invariant clauses needed to prove other proof obligations.

6 Dealing with Multiple Clients

In this section we show how to generalise the gluing invariant for a single client to deal with any number of concurrent clients.

6.1 Alterations on both B Machines

In the machine and refinement the variables associated to the client where extended to partial functions, where the domain of those functions describe the clients currently on-line.

\[
\begin{array}{|l|}
\hline
\text{MACHINE } \text{Bookstore} 0_n \\
\text{SETS} \\
\text{CLIENT} \\
\text{BOOK} \\
\text{STATE} = \{ a_1, a_2, a_3, a_4, a_5 \} \\
\text{INARIANT} \\
\text{basket} \in \text{CLIENT} \Rightarrow F(\text{BOOK}) \land \\
\text{budget} \in \text{CLIENT} \Rightarrow \mathbb{N}_1 \land \\
\text{accepted} \in \text{CLIENT} \Rightarrow \text{BOOL} \land \\
\text{state} \in \text{CLIENT} \Rightarrow \text{STATE} \land \\
\text{shelf} \in \text{BOOK} \Rightarrow \mathbb{N} \land \\
\text{price} \in \text{BOOK} \Rightarrow \mathbb{N}_1 \\
\hline
\end{array}
\]

\[
\begin{array}{|l|}
\hline
\text{REFINEMENT } \text{Bookstore} 1_n \\
\text{REFINES } \text{Bookstore} 0_n \\
\text{SETS} \\
\text{STATE} = \{ c_1, c_2, \ldots, c_{10}, c_{11} \} \\
\text{INDEX} = \{ i, i1 \} \\
\text{INARIANT} \\
\text{basket} \in \text{CLIENT} \Rightarrow F(\text{BOOK}) \land \\
\text{budget} \in \text{CLIENT} \Rightarrow \mathbb{N}_1 \land \\
\text{accepted} \in \text{CLIENT} \Rightarrow \text{BOOL} \land \\
\text{state} \in \text{CLIENT} \Rightarrow \text{STATE} \land \\
\text{shelf} \in \text{BOOK} \Rightarrow \text{seq(BOOK)} \land \\
\text{price} \in \text{BOOK} \Rightarrow \mathbb{N} \land \\
\text{price} \in \text{BOOK} \Rightarrow \mathbb{N}_1 \\
\hline
\end{array}
\]

Each operation of the abstract and concrete system will have an extra parameter, the client that is invoking the operation.

6.2 Alterations on the Abstraction Invariant

The gluing invariant \( I_B \) for the bookstore refinement will be similar to the client invariant \( I_C \). Although the invariant \( I_B \) needs an extra clause asserting that the domains of the variables related to the bookstore clients are the same. The last conjunction of the clause \( B_0 \) states that the set of clients on-line in the abstract and concrete system must be the same.

\[
B_0: \text{dom(basket1)} = \text{dom(budget1)} = \text{dom(accepted1)} = \text{dom(state1)} = \text{dom(thread1)} \\
\land \text{dom(basket1)} = \text{dom(basket)}
\]

As we said \( I_B \) is similar to \( I_C \) and, with the exception of \( B_0 \), all the clauses of \( I_B \) where obtained by generalising each \( I_C \) clause for a set of clients. The gluing invariant \( I_B \) is defined as the following conjunction:

\[
I_B = B_0 \land C_1 \land C_2 \land \cdots \land C_7
\]
The clause $C_1$ stays unaltered, generalising it over a set of clients does not alter the original clause. We will describe in more detail the clauses $B_{4.1}$ and $B_7$.

Clause $B_{4.1}$ universally quantify clause $C_{4.1}$ over the set of on-line clients. A client is on-line if it is defined for the any of the client’s partial functions. In $B_{4.1}$ we used function $\text{state1}$.

\[
B_{4.1} \forall \text{client} . \text{client} \in \text{dom(state1)} \land \\
\text{state1(client)} \in \{c7, c8\} \Rightarrow \text{basket(client)} = \emptyset
\]

Clause $B_7$ states the same property of clause $C_7$, that a book can either in the shelf or in the basket, although in the former clause one has to consider that a book might be in baskets of several clients.

\[
B_7 \forall \text{book} . \text{book} \in \text{BOOK} \Rightarrow \\
\text{shelf(book)} + \text{booksSold(book, basket)} = \\
\text{shelf1(book)} + \text{booksSold(book, basket1)}
\]

6.3 Proving the Refinement

The fact that almost every clause of the invariant has a universal quantification increased considerably the complexity of the proof obligations. The total number of proofs amounts to 250 and only 93 where automatically proved by the Atelier-B prover. From the remaining 157 proofs, 100 of those were time consuming but not difficult because we replicated the strategies used in the refinement of the single client system. Of the other 57 proofs, 45 where fairly difficult, and the last 12 proof obligations where extensive and difficult to prove. Nevertheless, all were proved.

7 Conclusions

In this paper we have showed how refinement can be used to analyse business processes with compensation. A model involving compensation may not give a clear view of the overall properties of the system. Moreover, as we have seen in the e-bookstore example, such a model may temporarily falsify those properties, and then use the compensation information to restore the system consistency.

We have explored an approach to the refinement of StAC specifications, where both StAC processes and B machine are refined simultaneously. This approach combines the both parts of a business process specification into a B machine that deals explicitly with compensation tasks and the ordering of operations. We have used Atelier-B to help constructing the refinement invariant, and also to prove the necessary proof obligations.

Our initial experience of applying our refinement approach to the bookstore example was that it is a difficult process. That view changed when we tried to prove the refinement of a single client, as most of the invariant clauses where constructed by the Atelier-B prover and it was possible to check informally if
the proof obligations where provable or not. We can conclude that the difficulty level of applying the StAC refinement strategy depends significantly on the system under study. The difficulty level was reasonable for the single client system, but too high for the bookstore system. The complexity in proving the invariant for the bookstore refinement was the result of the universal quantification on the invariant clauses. Those clauses where necessary because the bookstore was defined as a generalised parallel process. Considering that for the case study presented here, the invariant for the refinement of a single process was easily extended to a set of concurrent processes, possibly this tactic could be used for other systems defined as generalised parallel processes.

Although the approach we used in employing Atelier-B to construct the refinement invariant was fairly systematic, some reasoning is needed to deduce new obligations required by the invariant. It has to be investigated whether this approach could be automated, as found, for example, the automatic invariant generation provided by STeP [4].

An alternative to refinement would be to develop a model checker for StAC processes. A model checker would allow the verification of several types of properties, as for example invariants and assertions. Within this line of work, the paper [3] describes the use of both SPIN and STeP to verify StAC processes.

References