Using the extensible model checker XTL to verify StAC Business Specifications

Juan C. Augusto, Michael Leuschel, Michael Butler, Carla Ferreira

Department of Electronics and Computer Science
University of Southampton,
e-mail: {jca, mal, mjb, cf}@ecs.soton.ac.uk

Abstract. StAC is a business specification language that has been developed as part of a partnership program between IBM UK Labs. and the University of Southampton. It is highly desirable for Business specification languages like StAC to have a way to explore the correctness of specifications. We provide some details about how XTL can provide model checking capabilities for StAC and why it proved to be more successful than SPIN and STeP in that sense.

1 Introduction

Because of their complexity, business transactions are prone to failure in many ways. For example, a request that normally is satisfied under certain conditions can be unexpectedly rejected. That can be experienced in daily life when the book we requested on the web is not anymore in stock, when our trip is cancelled, or when an appointment we scheduled cannot be made effective.

However systems are normally built considering the normal and expected pattern of behavior. A way to deal with this conflict is to supplement the usual pattern of behavior with mechanisms which allow the system to react more appropriately when an unexpected/undesired event occurs. One such mechanism proposed in the literature is to associate a *compensation* action to each action, which will repair or handle in an appropriate way abnormal situations. Offering alternatives and rescheduling can be ways to compensate previous actions. Recently, IBM, Microsoft, and BEA have been developing a business process language called BPEL [CGK+] which provides a compensation notion. Because this language lacks a formal semantics we focus instead on StAC (see [BF00], [CGV+02], and [Fer03]), another business process modelling language which formally handles compensation.

Previous attempts to allow automatic verification of StAC specifications involved the exploration of well-known tools like SPIN [Hol97] and STteP [BBC+99]. The first option led us to consider a translation from the StAC specification language to Promela, the input language for SPIN. For the second option we considered instead a translation to SPL, the input language for STeP. We faced several problems [AB02] preventing us from obtaining an easy to use verification framework by using any of these tools. This paper gives details about a more

succesful experience by using XTL (see [LM00] and [LM02]) a model checker that takes a more high level input language than SPIN and STeP and provides model checking facilities for properties written in CTL [CES86].

After an introduction to StAC (section 2) we provide some brief explanation about why model checking StAC with SPIN and STeP (section 3). Then (section 4) we introduce XTL and provide samples of properties that can be model checked by using XTL. The conclusions (section 5) will summarize some of the lessons learnt during all the three attempts to verify StAC specifications.

2 StAC

StAC (Structured Activity Compensation) is a language that, in addition to CSP-like operators [Hoa85], offers a set of operators to handle the notion of *compensation*. In StAC it is possible to associate to an action a set of compensation actions providing a way to repair an undesired situation. Compensations are expressed as pairs with the form $P \div Q$, meaning that Q is the compensation planned in case that the effect of P needs to be compensated at a later stage. As the system evolves, compensations are remembered. If all the activities are successfully accomplished then the operator *accept*, $\boxtimes$, releases the compensations. If any activity fails then the operator *reverse*, $\boxtimes$, orders the system to apply all the recorded compensations for the current scope. In some contexts the failure to accomplish an activity can be so critical that demands the abortion of a process, that is the role of the *early termination* operator. Both compensation and termination operators are bounded to a scope of application.

DEFINITION **1** Let A represent an activity, b a boolean condition, P and Q two generic processes, x a variable and X a set of values. Then, we can define as follows the set of well formed formulas in StAC:

Process ::=	A	(activity label)		
|	0	(skip)	| $b \rightarrow P$	(condition)
|	*rec(P)*	(recursion)	| $P;Q$	(sequence)
|	$P \| \| Q$	(parallel)	| $\| \| x \in X.P_x$	(generalised parallel)
|	$P [] Q$	(choice)	| $[] x \in X.P_x$	(generalised choice)
|	$\odot$	(early termination)	| $\{P\}$	(termination scoping)
|	$P \div Q$	(compensation pair)	| $[P]$	(compensation scoping)
|	$\boxtimes$	(reverse)	| $\boxtimes$	(accept) ▲

In the examples below, processes written in boldface are intended to be basic activities. Each StAC specification is coupled with a B machine [Abr96] describing the state of the system and its basic activities. Basically a B machine is composed by a declaration of sets, variables, invariants, initialisation and operations over those structures. Each StAC activity in a specification will have associated an operation in the corresponding B machine explaining how that activity is implemented in logical terms. We address the reader who want a more in-detail account of StAC to [CGV+02] and [BF00].

EXAMPLE **1** First we consider the specification of an order fulfillment scenario presented in [CGV+02]. The whole process can be described throughout the following steps: a) an order is accepted from a customer; b) the warehouse prepares the order for shipment, including booking a courier for delivery; c) simultaneously with step (b) there is a credit check to verify if the customer can pay the order; d) if the check is successful the order completes, otherwise it is stopped and the compensation mechanism is started. This can, for example, send a request for unpacking the ordered items. In this case we consider three items.

```
abc = (acceptOrder ÷ restockOrder);
      fulfillOrder;
      ((okFulfillOrder → ☑ ) □(notokFulfillOrder → ☒ ))

fulfillOrder = {wareHousePackaging ||
                 (creditCheck ;
                 ((notokCreditCheck→ ⊙)□(okCreditCheck → 0))) }

wareHousePackaging = (bookCourier ÷ cancelCourier) || packOrder

packOrder = || i∈I .(packItem(i)÷ unpackItem(i))
```

△

EXAMPLE **2** Lets consider the specification of an E-Bookstore, presented in [BF00]. Here we consider a finite set C of registered customers. `Arrive` creates and initialises the client information, that includes setting the appropriate budget. `chooseBooks` represents the selection of books by the customer. Next, the customer is offered either to `quit` the transaction or to proceed by buying the chosen items. In the first option the information is cleared just after the compensation is applied (returning the books to the shelf), and in the second case it will be cleared after accepting the operation. The recursive process chooseBooks, allows the customer to choose as many books as s/he wants. For each book the customer chooses, it is checked if that lead to exceed her/his budget. If so, then the reverse operator causes the book to be returned to the shelf. `Pay` has the same effect in case checking the credit card is not successful.

```
Bookstore = || c∈C . client(c)

client(c) = arrive(c); chooseBooks(c);
            ((quit(c) ; ☒ ) □ (pay(c); ☑ )); exit(c)

chooseBooks(c) = checkout(c) □ (chooseBook(c); chooseBooks(c))

chooseBook(c) = □b∈B . [(addBook(c,b) ÷ returnBook(c,b)) ;
                        overBudget(c) → ☒ ]

pay(c) = processCard(c) ; accepted(c) → ☒
```

△

3 Previous Attempts

We provide here a brief account of the different problems we faced in our previous attempts to provide verification facilities for StAC. The reader can see all the details in [AB02].

3.1 Translating StAC to SPIN

Model checking can be used to check whether the specification of a system is consistent with a logical property. A particularly successful implementation of this approach is SPIN, [Hol97] that has been widely accepted as a tool for verification of software and hardware specifications. SPIN offers the possibility to perform simulations and verifications. Through these two modalities the verifier can detect absence of deadlocks and unexecutable code, to check correctness of system invariants, to find non-progress executions cycles and to verify correctness properties expressed in propositional linear temporal logic formulae. Promela is the specification language of SPIN. It is a C-like language enriched with a set of primitives allowing the creation and synchronization of processes, including the possibility to use both synchronous and asynchronous communication channels.

We refer the reader to the extensive literature about the subject as well as the documentation of the system at Bell Labs web site for more details: `http://netlib.bell-labs.com/netlib/spin/whatispin.html` We assume some degree of familiarity with this framework from now on.

Translating StAC specifications to Promela proved to be a non-trivial matter and, when possible, demanded much more complex data and control structures to recreate StAC distinctive features.

Coordinating Nested Procedures in Promela to mimic StAC procedure-like structure is one of the problems that can be solved but at the cost of introducing some coordinating mechanisms that will cause computational extra cost. For example, sequential calls to non-primitive processes in StAC behave as calls to procedures in programming languages. For example, a sequence of calls to non-primitive processes in the StAC specification must be executed without interleaving between them, while proctypes in Promela will allow interleaving. For example, "`run P; run Q`" will start `P` first and then will start `Q` without waiting for `P` to terminate. `Q` can be started at any time after `P` has been. The `;` operator in this case does not have the usual semantics for procedures in high level programming languages as it is the case for StAC. Synchronization can be achieved as expected in StAC through a fork & join mechanism forcing all subprocesses to be finished before the process that created them is considered finished. But, that is achieved at the price of adding channels and hence enlarging the search space at the time of real specifications. Other StAC operators like *generalised parallel* and *generalised choice* pose similar coordination problems with the addition that they are not provided as primitives in Promela, forcing to high level simulations.

The use of *generalised parallel* and *generalised choice* demands the enumeration of values for the parameters involved. While these parameters can be strictly

numeric users using StAC for real applications will be interested in using realistic labels for naming people, books, or hotels as in the examples shown in section 2 without being concerned by on mappings from names to numbers or other levels of indirection. Here we have another clash between the different levels of abstraction provided by Promela and StAC. The limitations to use enumerates in Promela are evident and altough it is possible to implement a more general use of enumerates, it demands an involved procedure which again obscures the initial specification and add more computational complexity at model checking time.

The *early termination* operator, $\odot$ (see example 1 for an illustration of its use), can be applied to force a process to terminate. Brackets can be used to delimitate the scope for the operator application. For example $\{P; \odot; Q\}; R$ specifies that after P is executed, Q will be forced to terminate. This will not affect R. If we apply $\odot$ to a parallel process then all the parallel processes within the scope of the $\odot$ are also terminated. For example, in $\{(P; \odot; Q)||R\}||S$ process R will also be terminated but S will not. We found that the implementation of this characteristic is particularly problematic in Promela. In Promela there is no direct way to encode this, the solution being again handling a parallel and high-level scheduling in order to force SPIN to terminate some of the processes involved.

Handling of compensations can be achieved by the use of a LIFO structure. Stored codes can be recovered later, if necessary, to know what compensations must be applied and in which order that must be done. Each possible compensation activity is identified with a code. In some cases more information can be stored, e.g., in example 2 we need to store a code to identify the operation of giving back books, but also what books are involved in the operation. Both, the complexity of the structure dictated by the kind of compensations we need for some case studies, and the need of the generalised parallel to inspect the structure are serious drawbacks in terms of search complexity, an important issue for finite-state verification. Then, we found that implementing the very basic operations related to compensation handling was also a major issue in terms of the computational complexity required.

3.2 Translating StAC to SPL

STeP ([BBC+99]) is a verification system for reactive systems based in a deductive approach. It provides a collection of tools allowing verification by deduction, sometimes with user interaction. Some of the basic features provided are: verification rules, verification diagrams, and automated support for proving verification conditions. Model checking is also available, and is a good complement to the deductive system providing counter examples to false properties. A system can be input to STeP as an SPL program or as a *Fair Transition System* [MP92]. Systems specified in either of these notations must be given to STeP as input files.

The syntax of SPL programs follows that of traditional imperative languages such as Pascal. In addition to the basic constructs found in these languages, SPL

supports nondeterminism by means of the selection statement '`or`' and parallel composition by means of the cooperation statement ||. Parallel processes can interact through shared variables such as semaphores, as well as by synchronous and asynchronous channels. Execution of parallel processes is assumed to proceed by interleaving. For the sake of space we address the reader interested in specifying systems using SPL syntax or Fair Transition System notation to [MtSg95], chapter 2. The specification language for temporal properties to be checked is Linear-Time Temporal Logic [MP92]. More documentation about the system, including tutorials, demos for specific parts of the system and case studies can be found in the web page for STeP (http://www-step.stanford.edu/).

Some of the obstacles we came across while repeating our previous attempt to achieve automatic translations for verification of StAC specifications are the following.

Because SPL offers more high-level features than Promela, we found it easier to map StAC to the first language than to the later. One remaining obstacle is the lack of recursive processes as "the parser just plugs in the bodies of procedures when it finds a procedure call" ([MtSg95], pp. 29). Then general recursion cannot be directly implemented as used in StAC but instead we were able to use an equivalent translation, e.g. a While-like loop. Naturally, with the limitation that it can only be used with tail-recursion cases.

An advantage of SPL in comparison with Promela is that it provides generalised parallel and generalised choice sentences. The bad news being that SPL does not allow non-numeric enumerate values to be used in generalised choice and parallel. Again we have to resort to encodings, mapping strings into numbers and using numbers as a metaphor of the real information with the same negative consequences of the previous step.

SPL does not provide any constructor that can help to implement the *early termination* operator so we face similar problems to model check StAC specifications using that operator.

There are also problems falling outside the scope of the input language. For example, interpreting a counterexample given by the model checker is a very involved process as the steps that caused the unexpected situation are described in terms of internal variables acting as indirect references to the user's structures. There seems to be no syntax description in any of the publicly available documentation for the system. This force users to have a deep knowledge of all the theoretical framework underlying the system in order to be able to understand a counterexample.

4 XTL (eXtensible Temporal Logic model checker)

The XTL model checker allows the user to model check a wide range of system specification, (see for example [LM00] and [LM02]) the only requirement being that the specification is made by using high level Prolog predicates describing how the system makes transitions between its different states. In this section we describe some basic aspects of XTL and exemplify how to use it to

model check StAC specifications. XTL has been implemented using XSB Prolog (http://xsb.sourceforge.net/). It is still an ongoing research project but as described in the articles cited above has been tested with several different type of specifications with encouraging performance and expressiveness indicators. A GUI including availability of several formula schemas and a translator from a more natural language like notation to a Prolog oriented CTL notation is being developed.

The input language for XTL is very simple and general allowing multiple systems to be "plugged" for model checking. There are two key predicates, `trans/3` and `prop/2` where:

`trans(A,S1,S2)`: action `A` allows a transition from state `S1` to state `S2` and
`prop(S,C)`: condition `C` holds in state `S`

EXAMPLE **3** Lets assume we have the following declaration:

```
trans(t,a,b). trans(t,b,a). trans(t,c,c).
prop(a,safe). prop(b,safe). prop(c,unsafe).
```

This lines specifies that there are transitions from `a` to `b`, viceversa, and from `c` to itself. Also states `a` and `b` are labelled as safe while state `c` is labelled as unsafe. △

Then the main task when trying to use XTL to model check specifications written in a language L is to write a translator from L to a set of `trans/3` and `prop/2` predicates that allows XTL to traverse the related Kripke structure, by using `trans/3` to create new possible worlds and `prop/2` to state the truths on each world.

We reproduce below some of the operational semantics rules and their corresponding encoding in Prolog as used by XTL. The reader may observe in the following semantics rules the presence of indexes associated with compensation-related operators, indexed compensation replaces the need for compensation scoping [BF00].

Rules 10 and 11 are for compensation pairs. The first one states that an evolution in the main process, P, does not affect the compensation task, Q, while the second adds the compensation process, Q, to the compensation function, C, when the primary task of a pair has completed. C is a function that for each task i returns the associated compensation process, $C(i)$. σ is used to represent a state in the evolution of the system. In the Prolog rules given below is represented by using the predicate `conf/2`. Prolog predicate `push_comp` in the translation for Rule 11 is storing the compensation task, Q, into the corresponding compensation stack.

$$(R10)\ \frac{(P,\, C,\, \sigma) \xrightarrow{B} (P',\, C',\, \sigma')}{(P \div_i Q,\, C,\, \sigma) \xrightarrow{B} (P' \div_i Q,\, C',\, \sigma')}$$

```
/* R10 as used in XTL */
stac_trans(conf(pair(P,Q,I),C), B, conf(pair(P1,Q,I),C1)) :-
            stac_trans(conf(P,C), B, conf(P1,C1)).
```

$$(R11)\ \frac{}{(null \div_i Q,\ C,\ \sigma) \xrightarrow{\div_i} (null,\ C[i := Q; C(i)],\ \sigma')}$$

```
/* R11 as used in XTL */
stac_trans(conf(pair(null,Q,I),C), pairT(I), conf(null,R)) :-
            push_comp(C,I,Q,R).
```

Rule 12 is for the *reverse* operator, where compensation i is executed. Prolog predicate `comp_seq` is used in the Prolog encoding of Rule 12 to apply the compensation. Prolog predicate `clear_comp` as mentioned in Rule 12 and Rule 13 release the compensations associated with i.

$$(R12)\ \frac{}{(\boxtimes_i,\ C,\ \sigma) \xrightarrow{\boxtimes_i} (C(i),\ C[i := null],\ \sigma)}$$

```
/* R12 as used in XTL */
stac_trans(conf(compensate(I),C), compensate(I), conf(P1,C1)) :-
            comp_seq(C,I,P1), clear_comp_1(C,I,C1).
```

Rule 13 is for the *accept* operator where the compensation task i is cleared:

$$(R13)\ \frac{}{(\boxdot_i,\ C,\ \sigma) \xrightarrow{\boxdot_i} (null,\ C[i := null],\ \sigma)}$$

```
/* R13 as used in XTL */
stac_trans(conf(commit(I),C), commit(I), conf(null,C1)) :-
            clear_comp_1(C,I,C1).
```

Once the behavior of the system has been specified as a set of Prolog predicates the tool allows the specification to be checked against properties written using the formal language of temporal logic CTL (Computation Tree Logic),

introduced by Clarke and Emerson in [CES86]. CTL allows to specify properties of specifications generally described as Kripke structures. The syntax for CTL is given below as a reminder and we address the reader to [CES86] and [CGP99] for more details about semantics and other subtle issues.

Given *Prop*, the set of propositions, the set of CTL formulae ϕ is inductively defined by the following grammar (where $p \in Prop$):

$$\phi := \ \mathbf{true} \mid p \mid (p) \mid \neg\phi \mid \phi \wedge \phi \mid \forall \bigcirc \phi \mid \exists \bigcirc \phi \mid \forall(\phi \mathcal{U} \phi) \mid \exists(\phi \mathcal{U} \phi)$$

Since they are often used, the following abbreviations are defined

- $\forall\Diamond\phi =_{def} \forall(\mathbf{true}\mathcal{U}\phi)$ i.e. for all paths, ϕ eventually holds,
- $\exists\Diamond\phi =_{def} \exists(\mathbf{true}\mathcal{U}\phi)$ i.e. there exists a path where ϕ eventually holds,
- $\exists\Box\phi =_{def} \neg\forall\Diamond(\neg\phi)$ i.e. there exists a path where ϕ always holds,
- $\forall\Box\phi =_{def} \neg\exists\Diamond(\neg\phi)$ i.e. for all paths ϕ always holds.

As the sessions samples given below are pasted from an ASCII-based version of the XTL model checker we provide below a mapping table between CTL formal symbols and their ASCII version counterparts:

$CTL:$	$\neg$	$\wedge$	$\vee$	$\exists$	$\forall$	$\Diamond$	$\Box$	$\bigcirc$	$\mathcal{U}$
ASCII XTL:	`not`	`and`	`or`	`e`	`a`	`f`	`g`	`n`	`u`

EXAMPLE **4** A CTL formula $\exists\Diamond(\forall(\phi_1 \mathcal{U} \phi_2))$ becomes: `ef(au(phi1,phi2))` △

EXAMPLE **5** A CTL formula:

$$\begin{array}{l} \exists\Diamond(\ \phi_1 \\ \quad \wedge \\ \quad \exists \bigcirc (\neg(\phi_2) \vee \phi_3) \\ \quad) \end{array}$$

will become: `ef(and(phi1, en(or(not(phi2),phi3))))` △

Some of the predicates available in XTL are:

`sat/3`, where `sat(State,Formula,Trace)` checks satisfiability of `Formula` from an arbitrary `State` and gives a `Trace`.

`check_enable/1`, where `check_enable(P)` allows to check if a proposition is reachable. It does that by using the following definition:

```
check_enable(P) :-
  start(X), sat(X,ef(e(act([P|_]))),Trace), print(Trace), nl.
```

`ce/2`, where `ce(List,Trace)` searchs for compensations applicable to the list of entities given in `List`. This predicate in StAC will check for applicability of the operator "compensate" by using the following predicate: `sat([],ef(e(compensate(List))),Trace)`

Following we offer some samples of queries to the specifications considered in section 2. Those specifications can be entered in ASCII format and translated to a set of Prolog equations that is used to feed the XTL model checker. We can consider two kind of traces: verbose or simplified. The first one consider all the details considered by the system while applying compensation operations, including the handling of stacks to keep record of the compensations considered. Below we provide traces in the simpler form, highlighting just basic activities and the application of compensation operators.

4.1 Sample session for *OrderFulfilment* Case Study

Observe we make the specification provided in section 2 finite by assuming that the generalized parallel operator used in the process `packOrder` considers a set `I` = { `item1, item2, item3` }.

"Is `bookCourier` *reachable?"* The following query shows that after the main process, `abc`, is started the activity `acceptOrder` can take place and at that state holds that `bookCourier` is enabled:

```
| ?-  check_enable(bookCourier).

[start(abc),act([acceptOrder]),holds(e(act([bookCourier])),
[e(act([bookCourier]))])]
```

"Are there any compensations applicable?" The following query allows us to obtain a more surprising and interesting result. After the first answer we ask the system to find other possible situation where compensations can be applied and reading the two witnesses offered by the system we can discover there are at least two ways to activate the compensation mechanism:

```
| ?- ce(C,Trace).

C = [1]
Trace = [start(abc),act([acceptOrder]),act([packItem,i1]),
act([packItem,i2]),act([bookCourier]),act([creditCheck]),
act([okCreditCheck]), act([notOkFulfillOrder]),
holds(e(compensate([1])),[e(compensate([1]))])];

C = [1]
Trace = [start(abc),act([acceptOrder]),act([packItem,i1]),
act([packItem,i2]),act([bookCourier]),act([creditCheck]),
act([notOkCreditCheck]),exit(1),act([notOkFulfillOrder]),
holds(e(compensate([1])),[e(compensate([1]))])]
```

The system as modelled in the StAC specification provided in section 2 is not supplemented with the B machines that will implement activities and their side effects. Then compensations will be applied disregarding if the credit card check

fails or not because the test about correctness of FulfillOrder that otherwise would be directly dependant on the credit card check is completely independent in this model. We can check this in a more general way. Lets assume we want to check that there are "No unmotivated courier cancellations" or, equally useful for us, lets see if we can produce a witness that there could be a courier cancellation despite the credit card was approved. A quick simplification, for example by using the following shorter specification:

```
abc = (acceptOrder ÷ restockOrder);
      wareHousePackaging;
      ((okCreditCheck → ☑ ) ▯(notokCreditCheck → ☒ ))

wareHousePackaging = (bookCourier ÷ cancelCourier) || packOrder

packOrder = || i∈I .(packItem(i)÷ unpackItem(i))
```

and running the query again will provide the expected answer, compensations occurs only when the credit card check fails:

```
| ?- ce(C,Trace).

C = [1]
Trace = [start(abc),act([acceptOrder]),act([bookCourier]),
act([packItem,i1]),act([packItem,i2]),act([packItem,i3]),
act([notokCreditCheck]), holds(e(compensate([1])),
[e(compensate([1]))])];
C = [1]
Trace = [start(abc),act([acceptOrder]),act([bookCourier]),
act([packItem,i1]),act([packItem,i2]),act([packItem,i3]),
act([notokCreditCheck]),holds(e(compensate([1])),
[e(compensate([1]))])];
```

4.2 Sample session for *EBookstore* Case Study

In this case study the set `C` in the process `Bookstore = || c∈C . client(c)` is assumed to be instantiated with two customers: `sofia` and `mel` and the set `B` in `chooseBook(c)` is assumed to have the following books: `prolog, java, xml`. It must be observed the infinite loop in the specification of process `chooseBooks(c)` can make the stack for compensations to grow indefinitely. To avoid that the initial specification has been limited to a maximum of three iterations by redefining that process as follows:

chooseBooks(c) = **checkout**(c) ▯ (chooseBook(c); chooseBooks1(c))
chooseBooks1(c) = **checkout**(c) ▯ (chooseBook(c); chooseBooks2(c))
chooseBooks2(c) = **checkout**(c) ▯ (chooseBook(c); chooseBooks3(c))
chooseBooks3(c) = **checkout**(c)

"Is `exit` *reachable?"* is answered by the system with a witness showing that a user can leave the system.

```
| ?- check_enable(exit).

[start(bookstore),act([arrive,sofia]),act([arrive,mel]),
act([checkout,mel]),act([quit,mel]),compensate([mel]),
holds(e(act([exit,mel])),[e(act([exit,mel]))])]
```

"Can be Sofia compensated?" is answered by the system showing a sequence of operations where the customer buy a book but then finally cannot finish the operation (when `overBudget` is activated) so the system returns the book to the stock.

```
| ?- ce([sofia],Trace).

Trace = [start(bookstore),act([arrive,sofia]),act([arrive,mel]),
act([checkout,mel]),act([quit,mel]),compensate([mel]),
act([exit,mel]),act([addBook,sofia,prolog]),act([overBudget,sofia]),
compensate([sofia]),act([returnBook,sofia,prolog]),
act([checkout,sofia]),act([quit,sofia]),
holds(e(compensate([sofia])),[e(compensate([sofia]))])]
```

"Is each customer forced to do a transaction?" is answered negatively by the system. Further exploration provide us with a possible scenario where a customer, Sofia, leaves the system without purchasing books:

```
| ?- start(X), sat(X,ag(imp(e(act([arrive,sofia])),
(ef(e(act([processCard,sofia])))))),T).
(no)

| ?- start(X), sat(X,ef(imp(e(act([arrive,sofia])),
(ef(and(not(e(act([processCard,sofia]))),e(act([exit,sofia])))))))),T).

X = []
T = [holds(imp(e(act([arrive,sofia])),
ef(and(not e(act([processCard,sofia])),e(act([exit,sofia]))))),
[start(bookstore),act([arrive,sofia]),act([arrive,mel]),
act([checkout,mel]),act([quit,mel]),compensate([mel]),
act([exit,mel]),act([checkout,sofia]),act([quit,sofia]),
compensate([sofia]), holds(and(not e(act([processCard,sofia])),
e(act([exit,sofia]))),[and([not e(act([processCard,sofia]))],
[e(act([exit,sofia]))])])])]
```

5 Conclusions and Further Work

Business transactions can be very complex and ensuring correctness is a critical issue. We focused on the problem of providing automatic verification for a business-related specification language, StAC, by using the XTL model checker. Its high-level specification language allowed us to overcome previous difficulties we faced in our previous attempts by using SPIN and STeP.

Many, but not all, of the problems we faced when using SPIN and STeP were about mapping a high level language as StAC to the control and data structures provided in Promela and SPL. In some cases the complexity of translation and space exploration of the resulting model check process increases up to an undesirable level and of course there is no possibility to tailor any of those tools to particular needs. In this case XTL (eXtensible Temporal Logic model checker) can be adapted to different input languages as illustrated in previous reports about the tool in the literature. In particular it can be successfully applied to StAC. It was far simpler (and possible) to map StAC to XTL than to either SPIN or STeP. As explained in section 4, the mapping from StAC operational semantics to XTL's input language is quite natural and straightforward compared with the involved procedure required to do the mapping to either Promela or STL.

Because of this flexibility in the input language, the encouraging performance results and the well-known language used to specify the properties to be checked (CTL) we believe XTL is a tool that colleagues working in different areas may find worth to investigate more closely.

References

[AB02] J. Augusto and Michael Butler. Some Observations About Using SPIN and STeP to Verify StAC Specifications. Technical report, 2002. Electronics and Computer Science Department, University of Southampton. 34 pages. http://www.ecs.soton.ac.uk/ jca/st2s.pdf.

[Abr96] J. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University, 1996.

[BBC+99] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, B. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A step tutorial. *Formal Methods in System Design*, 16:227–270, 1999.

[BF00] M. Butler and C. Ferreira. A process compensation language. In *IFM'2000 - Integrated Formal Methods, volume 1945 of LNCS*, pages 61–76. Springer Verlag, 2000.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGK+] Francisco Curbera, Yaron Goland, Johannes Klein, Frank Leymann, Dieter Roller, Satish Thatte, and Sanjiva Weerawarana. Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/library/ws-bpel/.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[CGV+02] M. Chessell, C. Griffin, D. Vines, M. Butler, C. Ferreira, and P. Henderson. Extending the concept of transaction compensation. *IBM Journal of Systems and Development*, 41(4):743–758, 2002.

[Fer03] C. Ferreira. Precise modelling of business processes with compensation. PhD Thesis, Electronics and Computer Science Department, University of Southampton, 2003.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol97] Gerard Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[LM00] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In *Proceedings of Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, pages 63–82. editor Annalisa Bossi, Venice, Italy, LNCS 1817, 2000.

[LM02] Michael Leuschel and Thierry Massart. Logic programming and partial deduction for the verification of reactive systems: An experimental evaluation. In *Proceedings of 2nd Workshop on Automated Verification of Critical Systems (AVOCS'02), Birmingham (UK)*, pages 143–150, 2002.

[LR00] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems (Specification)*. Springer Verlag, 1992.

[MtSg95] Zohar Manna and the STeP group. STeP: The Stanford Temporal Prover (Educational Release), User's Manual. Technical report, 1995. STAN-CS-TR-95-1562, Computer Science Department, Stanford University. 138 pages.