Using SPIN and STeP to Verify Business Processes Specifications

Juan C. Augusto Michael Butler Carla Ferreira Stephen Craig

Declarative Systems and Software Engineering Research Group
Department of Electronics and Computer Science
University of Southampton, Southampton, UK
{jca, mjb, cf, sjc02r}@ecs.soton.ac.uk

Abstract. Business transactions are prone to failure and having to deal with unexpected situations. Some business process specification languages, e.g. StAC, introduce notions like *compensation* handling. Given the need of verification of correctness in business related software, it is important to fill in the gap between business process specification languages like StAC and the verification software already available.
We report on two of our previous attempts to develop a tool to allow verification of StAC specifications by using already existing systems, SPIN and STeP. We highlight some of the problems we faced during these attempts as they can prevent successful and widespread use of verification tools. Our experience can be used to make the available tools more versatile and hence, useful to a wider range of applications.

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 is not anymore in stock, or when our trip is cancelled.

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. We focus on StAC (see [5] and [4]), a business process modelling language with a formal semantics which handles compensation.

We report here some of our attempts to provide a suitable verification framework for StAC specifications. We considered two systems with different characteristics, SPIN [7] and SteP [3]. 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.

This paper is an abstract of a larger article [1] which gives more details about the main contribution. After a brief introduction to StAC (section 2) we explain some of the main obstacles we found attempting to use SPIN (section 3) and STeP (section 4) to verify StAC specifications. We also provide a description on how we handled compensations (section 5) and samples of the verifications we were able to make (section 6). The conclusions (section 7) will summarize some of the features that need to be made available in the next generation of verification systems for business-related systems.

2 StAC

StAC (Structured Activity Compensation) is a language that, in addition to CSP-like operators [6], offers a set of operators to handle the notion of *compensation*. In StAC 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. Each compensation operator is bounded to a scope of application. If all the activities are successfully accomplished then the operator *accept*, ☑ , releases the compensations. If any activity fails then the operator *reverse*, ☒ , orders the system to apply all the recorded compensations for the current scope. The abortion of a process can be imposed by using the *early termination* operator, $\odot$.

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)
|	☒	(reverse)	| ☑	(accept) ▲

In the examples below, processes written in boldface are intended to be basic activities. Each StAC specification is coupled with a B machine [2] describing the state of the system and its basic activities. We address the reader who want a more in-detail account of StAC to [5] and [4].

Example **1** (*order fulfillment scenario* [5]) 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.

```
abc = (acceptOrder ÷ restockOrder);
      fulfillOrder;
      ((okFulfillOrder → ☑ ) [](notokFulfillOrder → ☒ ))
fulfillOrder = {wareHousePackaging ||
                 (creditCheck ;
                 ((notokCreditCheck→ ⊙)[](okCreditCheck → 0))) }
wareHousePackaging = (bookCourier ÷ cancelCourier) || packOrder
packOrder = || i∈I .(packItem(i)÷ unpackItem(i))
                                                                     △
```

3 Translating StAC to Promela

Model checking can be used to check whether a logical property is consistent with the specification of a system. A particularly successful implementation of this approach is SPIN, [7] that has been widely accepted as a tool for verification of software specifications. 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 more complex data and control structures to recreate StAC novel features.

Coordinating Nested Procedures. 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 their proctype counterparts 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 expected 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. Broadly speaking, a way to introduce this mechanism in the translation, e.g. by using channels, is as follows: for every process P calling other subprocesses $p_1, \ldots, p_n$, i.e. implemented as `proctype` calls through the `run` sentence, we can a) add at the end of each p_i, with $i = 1, \ldots n$, a way to acknowledge that the process have finished, and b) after the call sentence in the caller process a way to recognize that the called processes finished before proceeding.

For parallel processes let us consider the general case: `A(...) || B(...)`. We will call the parallel definition to be coordinated *a block*. A block `A || B` terminates when both `A` and `B` terminates. Differently from the sequential case we want to ensure that parallelism is restricted to those processes in the block. In this case the testing for acknowledgments is shifted immediately after the translation of the intervening processes inside the block.

If we use a parallel or a generalised parallel operator, we want to ensure that we consider the process finished if and only if all the processes being run concurrently are finished. Then again we need to address the coordination problem. Since Promela does not support generalised parallel we need to use a loop to create the appropriate number of parallel processes. Naturally the problem is far simpler when using *generalised choice* because when the choice involves a procedure call all we need to check is that one call was made.

StAC allows recursive definitions. In this particular case, we cannot adapt the idea of using channels as before. Messages to ensure termination will successfully ensure all the calls ended before proceeding to execute code after the recursive call but it could be the case that messages generated for an instance $i1$ of the recursive proctype will be allowing to finish an instance $i2, i1 \neq i2$. One option would be to generate "keys", which univocally tie each acknowledge with a call. Another simpler, but partial, solution to the problem is to translate tail-recursive specifications to an iterative one.

Enumerates. A problem that applies to both, *generalised parallel* and *generalised choice*, is that the above schema assumes the indexes of the generalised operators are numeric. But, the usual case is to provide different enumerates for each operator, representing names, brands, addresses and all sort of useful labels motivated by real life applications. So, if a more flexible set of values has to be used, the limitations imposed by Promela's restriction to define all enumerates by using just one `mtype` are obvious. Although it is possible, see [1], to overcome the restrictions imposed in Promela to the use of enumerates that makes the specification unnecessary complicated and inefficient.

Early Termination. 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.

The closest approach to a solution was the use of the label `provided` available in Promela which impose conditions to the executability of a proctype, provided some conditions are fulfilled. Unfortunately a note in Promela user's manual, `"provided clauses are incompatible with partial order reduction"`, deters us to do so.

4 Translating StAC to SPL

STeP ([3]) is a verification system for reactive systems based in a deductive approach. STeP provides a collection of tools allowing verification by deduction, sometimes with user interaction. 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* [8].

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. The specification language for temporal properties to be checked is Linear-Time Temporal Logic [8]. 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/). Now we shortly describe some of the obstacles we faced translating StAC to SPL.

Recursive Specifications Because "the parser just plugs in the bodies of procedures when it finds a procedure call" ([9], pp. 29), general recursion cannot be used as needed in StAC. To overcome that we have to resort to an equivalent translation, e.g. a While-like loop like we used for our translation to Promela. Naturally, with same limitations, i.e. it can be used just with tail-recursion cases.

Flexibility on Generalised Operators An advantage of SPL in comparison with Promela is that provides generalised parallel and generalised choice sentences. The bad news being that SPL does not allow to use non-numeric enumerate values in generalised choice and parallel. For example, we cannot use: `[or O[c:[java..xml]] :: ...`. 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.

Early Termination SPL does not provide any constructor that can help to implement the *early termination* operator. There is nothing in SPL syntax similar to the label `provided` available in Promela to impose conditions on the executability of a process.

This forced us to implement that notion by using special structures and procedures. To detect when a concurrent process has finished we use a structure where to store inter-related processes and conditions of termination have to be inserted inside inter-related processes to make their executions threads dependent on each others computations.

Obtaining Counterexamples 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.

5 Handling of Compensations.

A FIFO structure is used to record compensations, as a result, when the compensation mechanism is applied the compensation will be executed following that strategy. As the stack used to implement the notion of compensation is made up of global structures, access to these structures should be coordinated amongst the various concurrent processes.

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. To grant that each compensation has a different code we need to force each generalised parallel or generalised choice affecting a compensation pair to generate a disjoint set of codes from those used in other compensation pairs.

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.

6 Verification

Because of all the problems mentioned before we were able to obtain fully automated verification just for a subset of StAC, e.g. excluding early termination, general recursion, and case studies demanding sophisticated use of enumerates in the case of the Promela specifications. The properties we verified by using SPIN and STeP were written in PLTL [8]. Some examples of properties we can verify by using SPIN in relation with the Fulfilment Order scenario follows. More case studies investigated can be found in [1].

No unmotivated courier cancellations: $\Box\neg$ (okCreditCheck $\wedge$ cancelCourier)

Each order will be packed: $\Box$ (acceptOrder $\rightarrow \Diamond$ (packorder $\vee\neg$ okCreditCheck))

No unwanted unpacks: $\Box$ (acceptOrder $\rightarrow$ ($\neg$ unpackItem $W\neg$ okCreditCheck))

7 Conclusions

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. We considered using two well-known tools in the literature of verification for hardware and software systems: SPIN and STeP. Although the goal of achieving full automatic support for StAC-based specifications was not completely achieved we have collected valuable experience from our research.

One of the results of our research is that we have have identified fragments of StAC that can be translated to Promela or SPL. Another very interesting outcome from our research was the detection of a list of features which are common to business-related specification languages and that are problematic to deal with, even for state of the art tools like SPIN and STeP. We provided in section 2 two case studies that are classic business-related specifications and cannot be satisfactorily mapped to either Promela or SPL.

Many, but not all, of the problems we faced are 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. We think reporting this kind of experiences is very valuable in order to stimulate and guide improvement of state of the art tools towards their next stage. Overcoming this limitations should be part of the agenda to make model checking and other verification frameworks accessible to a broader class of problems.

References

[1] J. Augusto and Michael Butler. Some Observations About Using SPIN and STeP to Verify StAC Specifications. Technical report, DSSE-TR-2002-9, 2002. Electronics and Computer Science Department, University of Southampton. 34 pages.

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

[3] 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.

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

[5] 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.

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

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

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

[9] 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.