SEFM 2010

CNR, Pisa, Italy 13 - 18 September 2010

Software Engineering and Formal Methods

POSTERS and TOOL DEMO Proceedings

Sponsored by:

IEEE computer society

IEEE

intecs
the Brainware company

8th International Conference on
Software Engineering
and
Formal Methods

SEFM 2010

Proceedings of the
Posters and Tool Demo Session

CNR, Pisa, Italy 13-18 September 2010

Sponsored by:



The papers comprised in this book reflect the presentations perfomed at the Posters and Tool Demo Session of the 8th International Conference on Software Engineering and Formal Methods (SEFM 2010).
Session Chairs and proceedings editors are Franco Mazzanti and Gianluca Trentanni.
The inclusion of a contribution in this publication does not necessarily constitute endorsement by the editors and Session Chairs.

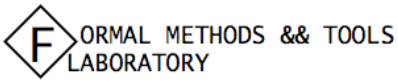
Online versions of these proceedings can be retrieved from:
http://puma.isti.cnr.it/search.php?langver=en

Published By
Consiglio Nazionale delle Ricerche
Area della Ricerca CNR di Pisa
Via Moruzzi, 1 - 56124 Pisa, Italy

ISBN 978-88-7958-006-9

ISTI Editorial 2010-ED-003

Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"



2010 Software Engineering and Formal Methods

SEFM 2010

Posters and Tool Demo Session

Table of Contents

Fastest: a Model-Based Testing Tool for the Z Notation

Maximiliano Cristiá
Flowgate Consulting and CIFASIS
Rosario – Argentina
mcristia@flowgate.net

Pablo Albertengo
Flowgate Consulting
Rosario – Argentina
palbertengo@flowgate.net

Pablo Rodríguez Monetti
Flowgate Consulting and UNR
Rosario – Argentina
prodriguez@flowgate.net

***Abstract*—Fastest is a model-based testing tool for the Z notation providing an almost automatic implementation of the Test Template Framework. The core of this document is an example showing how to use Fastest to automatically derive abstract test cases from a Z specification.**

I. Model-Based Testing

Model-based testing (MBT) is a well-known technique aimed to test software systems from a formal model [1], [2]. MBT approaches start with a formal model or specification of the software, from which test cases are generated, as shown in Fig. 1. These techniques have been developed and applied to models written in different formal notations, such as Z [3], finite state machines and their extensions [4], B [5], algebraic specifications [6], and so on.

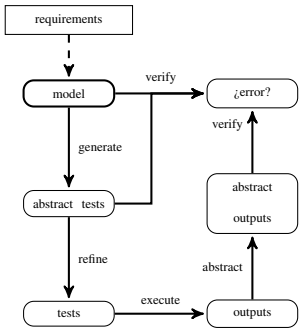


Figure 1: The MBT process.

The fundamental hypothesis behind MBT is that, as a program is correct if it verifies its specification, then the specification is an excellent source of test cases. As shown in Fig. 1, once test cases are derived from the model, they are refined to the level of the implementation language and executed. The resulting output is then abstracted to the level of the specification language, and the model is used again to verify if the test case has detected an error.

II. The Test Template Framework

The Test Template Framework (TTF) described by [3] is a particular MBT theory specially well suited for unit testing. The TTF uses Z specifications [7] as the entry models. Each operation within the specification is analysed to derive or generate abstract test cases. This analysis consists of the following steps:

1) Consider the valid input space (VIS) of each Z operation.
2) Apply one or more testing tactics in order to partition the input space.
3) Generate test objectives (specifications).
4) Prune inconsistent test objectives.
5) Find one abstract test case from each remaining test objective.

One of the main advantages of the TTF is that all of these concepts are expressed in the same notation of the specification, i.e. the Z notation. Hence, the engineer has to know only one notation to perform the analysis down to the generation of abstract test cases.

III. Fastest

Fastest implements the TTF allowing users to automatically produce test cases for a given Z specification. In [8] we show how Fastest semi-automates steps 1, 2 and 4 of the previous list; on the other hand, SEFM 2010 has accepted another paper written by us where we show how step 3 has been semi-automated too[1]. As far as we know, Fastest is the first MBT tool for the Z notation; surely it is the first one in implementing the TTF.

As is noted in [5], the TTF method was "quite widely known and referenced since" its first international publication. However, perhaps its lack of tool support made the MBT community to lose interest in it. Actually, the TTF's original authors only implemented Tinman [9] but, as Legeard in his colleagues say, "it was aimed primarily at providing organization support with little support for manipulating predicates". In effect, these authors conclude, after applying the TTF to a case study, that "TTF generation is a manual process, requiring extensive expertise at manipulating and simplifying schemas." Moreover, they found that the Z/EVES theorem prover [10] was not useful at performing those tasks since "in this case study the predicates were too difficult for the automatic simplification commands of Z/EVES." Finally, Legeard, Peureux and Utting conclude,

[1]The accepted paper is "Pruning Testing Trees in the Test Template Framework by Detecting Mathematical Contradictions".

among other things, that "a higher level of automation of the reasoning support would be useful" for the TTF, and that "BTT[2] is better designed for automation than TTF".

Our intention is to use this tool demonstration to show that these conclusions are at least doubtful.

IV. Architecture and Technology

Fastest is a Java application that should run on any platform running Java 1.6 or higher. It is based on the Community Z Tools (CZT) project [12], thus it reads Z specifications compliant with the Z ISO standard [13].

The tool was envisioned as a client-server application. The main reason for thinking of a distributed system came from the realization that calculating abstract test cases from test objectives in large projects could be a hard computing problem, but highly parallelizable as well. Then, we thought of an scalable application using the idle computer power present in a corporate network. A typical Fastest installation, thus, has some client processes and some testing servers.

Users interact with the application through the clients. The user interface of the client software is text-based, similar to command-line applications like Linux's bash, from which users can issue commands. Fastest asks each testing server to calculate a test case for a particular test class. Then, the time to find abstract test cases decreases proportionally with the number of available testing servers. This parallelization is so efficient because each calculation is completely independent from each other; synchronization is only needed when testing servers communicate a result to a client.

V. An Example

In this section we show how to use Fastest to derive abstract test cases for the Z specification described in Fig. 2. The specification is about the behaviour of the savings accounts system of simple bank. Table I summarizes the meaning of each basic type, state variable and operation. We think that this table plus the common knowledge about savings accounts will suffice to understand the model—we assume the reader is familiar with the Z notation.

The goal is, thus, to automatically derive abstract test cases—i.e. test cases written in Z—from this model by using Fastest. These abstract test cases can then be refined into a programming language—but this is out of the scope of this demonstration. As we have said, Fastest implements the TTF so we must follow it to derive abstract test cases. The next sections roughly follows the steps of the TTF as implemented by Fastest. Please, see the user manual for more details.

A. Writing the Specification

The specification must be written in the standard Z LaTeX mark-up [13] using any text editor. We suggest, however, to use Eclipse [14] with the CZT [15] and TeXlipse [16] plugins.

[2]BTT is a similar method for the B notation [11].

B. Launching Fastest

Fastest is executed from a command line issuing the following command from the installation directory:

```
java -jar fastest.jar
```

As we have said, the tool features a text-based user interface which prints a prompt and waits for commands:

```
Fastest>
```

C. Loading the Specification and Selecting Operations

Assuming the LaTeX file containing the specification is located in the installation directory, the specification is loaded with:

```
loadspec bank.tex
```

Once the specification has been successfully loaded the user has to select those schemas that represent the operations to be tested. Each operation is selected by entering a command like this:

```
selop NewClient
```

We want to test, also, the following operations:

```
selop NewAccount
selop Deposit
selop Withdraw
selop CheckBalance
selop AddOwner
```

D. Adding Testing Tactics

Testing tactics are the means proposed by the TTF to partition the VIS of a given operation. The more testing tactics added to an operation the more abstract test cases will be generated. However, it is not only a matter of adding many testing tactics but, better, the most promising ones for each operation. Fastest adds to any operation a testing tactic named DNF [8]. DNF is the first tactic to be applied. The user can add other tactics with, for instance:

```
addtactic NewClient SP \cup
          clients \cup \{u? \mapsto name?\}
```

In this example we add the following tactics, although they might not be the best choice, since we just want to make a demonstration of Fastest's capabilities.

```
addtactic NewAccount SP \notin
          n? \notin \dom balances
addtactic Deposit SP \oplus
          balances \oplus
          \{n? \mapsto balances~n? + m?\}
addtactic Withdraw NR m?
          \langle 10, 1000, 1000000\rangle
addtactic CheckBalance SP \in
          u? \mapsto n? \in owners
```

$[ACCNUM, UID, NAME]$
$MONEY == \mathbb{N}$
$BALANCE == \mathbb{N}$

Bank
$clients : UID \pfun NAME$
$balances : ACCNUM \pfun BALANCE$
$owners : UID \rel ACCNUM$

NewClientOk
$\Delta Bank$
$u? : UID;\ name? : NAME;\ n? : ACCNUM$

$u? \notin \dom clients$
$n? \notin \dom balances$
$clients' = clients \cup \{u? \mapsto name?\}$
$balances' = balances \cup \{n? \mapsto 0\}$
$owners' = owners \cup \{u? \mapsto n?\}$

$ClientAlreadyExists == [\Xi Bank;\ u? : UID \mid u? \in \dom clients]$
$AccountAlreadyExists == [\Xi Bank;\ n? : ACCNUM \mid n? \in \dom balances]$
$NewClient ==$
$\quad NewClientOk \lor$
$\quad ClientAlreadyExists \lor AccountAlreadyExists$

NewAccountOk
$\Delta Bank$
$u? : UID;\ n? : ACCNUM$

$u? \in \dom clients$
$n? \notin \dom balances$
$balances' = balances \cup \{n? \mapsto 0\}$
$owners' = owners \cup \{u? \mapsto n?\}$
$clients' = clients$

$ClientNotExists == [\Xi Bank;\ u? : UID \mid u? \notin \dom clients]$
$NewAccount ==$
$\quad NewAccountOk \lor$
$\quad ClientNotExists \lor AccountAlreadyExists$

DepositOk
$\Delta Bank$
$n? : ACCNUM;\ m? : MONEY$

$n? \in \dom balances$
$m? > 0$
$balances' = balances \oplus \{n? \mapsto balances\, n? + m?\}$
$clients' = clients$
$owners' = owners$

$AccountNotExists ==$
$\quad [\Xi Bank;\ n? : ACCNUM \mid n? \notin \dom balances]$
$IncorrectAmount == [\Xi Bank;\ m? : MONEY \mid m? \leq 0]$
$Deposit ==$
$\quad DepositOk \lor AccountNotExists \lor IncorrectAmount$

WithdrawOk
$\Delta Bank$
$u? : UID;\ n? : ACCNUM;\ m? : MONEY$

$u? \mapsto n? \in owners$
$n? \in \dom balances$
$m? > 0$
$m? \leq balances\, n?$
$balances' = balances \oplus \{n? \mapsto balances\, n? - m?\}$
$clients' = clients$
$owners' = owners$

$NotAnOwner ==$
$\quad [\Xi Bank;\ u? : UID;\ n? : ACCNUM \mid$
$\quad\quad u? \mapsto n? \notin owners]$

$InsufficientFunds ==$
$\quad [\Xi Bank;\ u? : UID;\ n? : ACCNUM;\ m? : MONEY \mid$
$\quad\quad m? > balances\, n?]$

$Withdraw ==$
$\quad WithdrawOk$
$\quad \lor AccountNotExists$
$\quad \lor IncorrectAmount$
$\quad \lor NotAnOwner \lor InsufficientFunds$

CheckBalanceOk
$\Xi Bank$
$u? : UID;\ n? : ACCNUM$
$balance! : MONEY$

$u? \mapsto n? \in owners$
$n? \in \dom balances$
$balance! = balances\, n?$

$CheckBalance ==$
$\quad CheckBalanceOk$
$\quad \lor AccountNotExists \lor IncorrectAmount$

AddOwnerOk
$\Delta Bank$
$u?, t? : UID;\ n? : ACCNUM$

$u? \mapsto n? \in owners$
$t? \mapsto n? \notin owners$
$owners' = owners \cup \{t? \mapsto n?\}$
$clients' = clients$
$balances' = balances$

$OwnerAlreadyExists ==$
$\quad [\Xi Bank;\ t? : UID;\ n? : ACCNUM \mid$
$\quad\quad t? \mapsto n? \in owners]$

$AddOwner ==$
$\quad AddOwnerOk \lor$
$\quad NotAnOwner \lor OwnerAlreadyExists$

Figure 2: A Z specification of the savings accounts of a banking system.

Term	Meaning
ACCNUM	The set of possible savings accounts numbers
UID	The set of identifiers of individuals (social security numbers, for instance)
NAME	The set of names of individuals
clients u	The name of person *u* as is recorded in the bank
balances n	The balance of savings account *n*
$owners(u,n)$	*u* is an owner of account *n*
$NewClient(u,name,n)$	Account *n* is opened by client *u* whose name is *name*
$NewAccount(u,n)$	Client *u* opens a new account with number *n*
$Deposit(n,m)$	The amount *m* is deposited in account *n*
$Withdraw(u,n,m)$	Client *u* withdraws amount *m* from account *n*
$CheckBalance(u,n,b)$	*b* is the balance of account *n* when client *u* checks it
$AddOwner(u,t,n)$	Client *u* adds *t* as an owner of account *n*

Table I: Meaning of the basic elements of the Z model of Fig. 2.

```
addtactic AddOwner SP \in
          u? \mapsto n? \in owners
addtactic AddOwner SP \notin
          t? \mapsto n? \notin owners
```

E. Generating Test Objectives

Test objectives—or specifications, classes or design—are automatically generated by running the following command:

```
genalltt
```

In doing so, Fastest performs a number of predicate manipulations as Legeard and his colleagues required in [5]. These objectives are structured as *testing trees*. Abstract test cases should be generated only from the leaves of these trees. In other words, each leaf stipulates some conditions under which the implementation must be tested.

F. Pruning Testing Trees

Although Fastest automatically generates test objectives, some of them may represent impossible situations. According to the TTF, each test objective is a set. Then, a test objective represents an impossible situation when its predicate is unsatisfiable. Unsatisfiable leafs should be pruned from testing trees. The automatic pruning strategy implemented in Fastest is executed with the following command:

```
prunett
```

This command, as `genalltt`, performs a series of predicate manipulations but of a different sort. The conception, design and implementation of this command is the subject of the paper published at this conference.

G. Deriving Abstract Test Cases

Deriving an abstract test case from a test objective means to find a vector of constant values satisfying the predicate of the objective. This task is performed with command:

```
genalltca
```

This is a much slower process compared to automatic pruning. When this command finishes some leaves have a child hanging from them representing the abstract test case. Each abstract test case is a Z schema box. The difference between a test objective and an abstract test case is that in the latter each input and state variable is bound to a constant value as exemplified in the following schema:

Deposit_SP_4_TCASE

Deposit_SP_4

$m? = 1$
$balances = \{(accnum0, 1)\}$
$clients = \emptyset$
$n? = accnum0$
$owners = \emptyset$

In this example, Fastest finds automatically all the abstract test cases but eleven. When this happens the user has to check whether the problematic objectives are satisfiable or not. In our case all but four are satisfiable. For those that are satisfiable, the user needs to help Fastest to find the required constants with a command like this one:

```
setfinitemodel CheckBalance_SP_5 -fm
   "owners==\{\{uid0 \mapsto accnum0\}\}"
```

For those that are not, the user has to add an *elimination theorem*; for instance:

ETheorem ArithmIneq4 [const $N, M : \mathbb{Z};\ n : \mathbb{Z}$]
$\mathsf{eval}(N \leq M)$
$n \leq N$
$M < n$

Elimination theorems are at the core of `prunett`; see the paper published by us at SEFM 2010.

H. Summary of Results

Table II summarizes the result of this experiment. As can be seen, Fastest generates 39 abstract test cases automatically from 46 possible test objectives. This high percentage is the consequence of two factors: (a) the high number of inconsistent test objectives removed by the automatic strategy, and (b) the heuristics implemented by `genalltca` to find constants values satisfying a given predicate. These

Operation	Leaves	Pruned Auto	Pruned Man	Remaining	ATC Auto	ATC Man
NewClient	24	15	0	9	9	0
NewAccount	6	1	0	5	5	0
Deposit	24	20	0	5	5	0
Withdraw	20	2	2	16	11	5
CheckBalance	6	0	0	6	4	2
AddOwner	12	8	0	5	5	0
Total	**92**	**46**	**2**	**46**	**39**	**7**
Total pruning time		**3.2s**	**Total ATC derivation time**		**38m40s**	

Table II: Summary of the results. Auto stands for automatically, Man for manually, and ATC for abstract test cases.

results are aligned with our previous experiments, see [8] and the paper to be published at SEFM 2010.

VI. Discussion

Fastest provides Z users with an almost automatic implementation of a sound MBT method originally thought for the Z notation. In this regard, Fastest overcomes all of the issues found by Legeard, Peureux and Utting in [5] making their last conclusion doubtful.

Fastest is freely available from [17], including a complete user manual in English. We have added entries to Wikipedia describing Fastest [18] and the TTF [19].

Perhaps there are many Z users out there thinking that MBT is an ideal technique for their daily software engineering work. Meanwhile, they watch other people using good MBT tools for their "younger" notations, but they found no implementation available for their beloved one. We hope Fastest could fill this gap.

References

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[2] R. M. Hierons and et al., "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 1–76, 2009.

[3] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777–793, Nov. 1996.

[4] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating finite state machines from abstract state machines," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2002, pp. 112–122.

[5] B. Legeard, F. Peureux, and M. Utting, "A Comparison of the BTT and TTF Test-Generation Methods," in *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*. London, UK: Springer-Verlag, 2002, pp. 309–329.

[6] G. Bernot, M. C. Gaudel, and B. Marre, "Software testing based on formal specifications: a theory and a tool," *Softw. Eng. J.*, vol. 6, no. 6, pp. 387–405, 1991.

[7] J. M. Spivey, *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[8] M. Cristiá and P. Rodríguez Monetti, "Implementing and applying the Stocks-Carrington framework for model-based testing," in *ICFEM*, ser. Lecture Notes in Computer Science, K. Breitman and A. Cavalcanti, Eds., vol. 5885. Springer, 2009, pp. 167–185.

[9] L. Murray, D. Carrington, I. Maccoll, and P. Strooper, "TinMan - A Test Derivation and Management Tool for Specification-Based Class Testing," in *In Technology of ObjectOriented Languages and Systems (TOOLS)*, 1999, pp. 222–233.

[10] M. Saaltink, "The Z/EVES System," in *ZUM '97: The Z Formal Specification Notation*, J. Bowen, M. Hinchey, and D. Till, Eds., 1997, pp. 72–85.

[11] L. B. P. F. Bouquet F., "Constraint logic programming with sets for animation of B formal specifications," in *CL'00 Workshop on (Constraint) Logic Programming and Software Engineering (LPSE'00)*, London, UK, 2000.

[12] CZT. Community Z Tools (CZT) project. [Online]. Available: http://czt.sourceforge.net

[13] ISO, "Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics," International Organization for Standardization, Tech. Rep. ISO/IEC 13568, 2002. [Online]. Available: http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip

[14] The Eclipse Foundation. Eclipse. [Online]. Available: http://www.eclipse.org/

[15] CZT. CZT Eclipse Plugin. [Online]. Available: http://www.cs.waikato.ac.nz/~marku/czt/eclipse.html

[16] T. Hupponen, K. Karlsson, J. Laitinen, O. Ojala, A. Pirinen, E. Seuranen, and L. Takkinen. TeXlipse. [Online]. Available: http://texlipse.sourceforge.net/

[17] Flowgate Consulting. Fastest. [Online]. Available: http://www.flowgate.net/?lang=en&seccion=herramientas

[18] ——. Fastest in wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Fastest

[19] ——. Test template framework in wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Test_Template_Framework

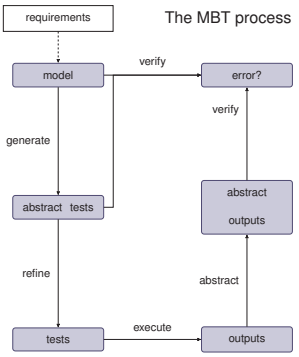
Fastest
A Model-Based Testing Tool for the Z Notation

Maximiliano Cristiá[12], Pablo Albertengo[1] and Pablo Rodríguez Monetti[13]

[1]Flowgate Consulting – [2]CIFASIS – [3]FCEIA-UNR
Rosario, Argentina

Model-Based Testing (MBT)

- MBT is a well-known technique aimed to test software systems from a formal model.
- It was implemented for some formal notations (B, FSM, etc.).
- Fastest is the first implementation for the Z notation.
- **We are showing how Fastest automatically produces test cases from a Z specification.**



The Test Template Framework

- Fastest implements a particular MBT method called Test Template Framework (TTF).
- TTF is concerned essentially with the "generation" step (see above).
- Within the TTF the "generation" step is as follows.

Steps of the TTF

1. Consider the input space of each Z operation.
2. Apply testing tactics in order to partition the input space.
3. Generate test objectives (specifications).
4. Prune inconsistent test objectives.
5. Find one abstract test case from each remaining test objective.

The Z Specification

Part of the specification of the savings accounts of a banking system—see the full specification in the accompanying document.

Bank

$clients : UID \pfun NAME$
$balances : ACCNUM \pfun BALANCE$
$owners : UID \rel ACCNUM$

NewClientOk

$\Delta Bank$
$u? : UID;\ name? : NAME;\ n? : ACCNUM$

$u? \notin \dom clients$
$n? \notin \dom balances$
$clients' = clients \cup \{u? \mapsto name?\}$
$balances' = balances \cup \{n? \mapsto 0\}$
$owners' = owners \cup \{u? \mapsto n?\}$

$ClientAlreadyExists == [\Xi Bank;\ u? : UID \mid u? \in \dom clients]$
$AccountAlreadyExists == [\Xi Bank;\ n? : ACCNUM \mid n? \in \dom balances]$
$NewClient == NewClientOk \lor ClientAlreadyExists \lor AccountAlreadyExists$

Operations not shown here:

NewAccount == ... an existing client opens another account ...
Deposit == ... somebody deposits money in an account ...
Withdraw == ... an owner withdraws money from one of his accounts ...
CheckBalance == ... an owner checks the balance of one of his accounts ...
AddOwner == ... an owner adds a friend as owner of one of his accounts ...

Writing the Specification

The specification must be written in the standard Z LaTeX mark-up using any text editor. We suggest, however, to use Eclipse with the CZT and TeXlipse plugins.

Launching Fastest

Fastest is executed from a command line as follows:

```
java -jar fastest.jar
```

The tool features a text-based user interface which prints a prompt and waits for commands:

```
Fastest>
```

Loading the Specification and Selecting Operations

The specification is loaded with:

```
loadspec bank.tex
```

Then the user has to select those schemas that represent the operations to be tested.

```
selop NewClient
```

Adding Testing Tactics

- Testing tactics are the means proposed by the TTF to partition the VIS of a given operation.
- Fastest adds automatically a testing tactic named DNF.
- The user can add other tactics by issuing a command like the one below:

```
addtactic NewClient SP \cup
          clients \cup \{u? \mapsto name?\}
```

Generating Test Objectives

Test objectives—or specifications, classes or design—are automatically generated by running the following command:

```
genalltt
```

Pruning Test Objectives

- Test objectives may represent impossible situations.
- Test objectives are sets.
- A test objective represents an impossible situation when its predicate is unsatisfiable.
- Run: `prunett`.

Finding Abstract Test Cases

Finding an abstract test case from a test objective means to find a vector of constant values satisfying the predicate of the objective. This task is performed with command:

```
genalltca
```

Summary

- Specification size: 139 lines of formal text, 6 operations, 3 state variables (complex)
- Test objectives: 48 unsatisfiable, 46 valid, 92 total.
- Pruning: 46 automatically in 3.2s, 2 manually
- Abstract test cases: 39 automatically in 38m40s, 7 manually

Dam Management Based on Model Checking Techniques

María-del-Mar Gallardo, Pedro Merino, Laura Panizo and David Sanán
Dept. Lenguajes y Ciencias de la Computación
University of Málaga, Málaga, Spain
Email: (gallardo,pedro,laurapanizo,sanan)@lcc.uma.es

Antonio Linares
BEFESA AGUA, SAU
Email: Antonio.linares@befesa.abengoa.com

***Abstract*—Dams are critical systems which, traditionally, have been managed manually and, more recently, by means of decision support systems based on numerical models and simulations. In this work, we describe how model checking techniques may be used to develop a tool oriented to dam management. The dam is modeled as a hybrid system to take into account the discrete and continuous aspects of its behavior. Although SPIN is not intended to analyze hybrid systems, the manner in which the continuous variables of the dam evolve (following difference equations) has allowed us to use it along with the PROMELA language to both describe the hybrid model and to carry out the necessary analysis leading to the construction of the automatic decision support tool.**

***Keywords*-Model checking, dam management, hybrid system.**

I. Introduction

Dams are critical systems that have been traditionally managed by human operators. Extreme and unexpected situations, such as flow periods, can lead to human errors. This is why the construction of tools to help operators in critical circumstances is crucial. In recent years, some decision support tools for basin management have emerged. Most of them are based on simulating complex numerical systems that provide the precise evolution of continuous magnitudes such as river flow. However, these tools have several inconveniences. For instance, one significant problem is that the techniques used are not suitable to manage discrete variables. Furthermore, simulations are very time consuming (the time needed to obtain results is around 2 hours.) Moreover, it is usually necessary to carry out more than one simulation to explore the effects of applying different dam management alternatives. Finally, these tools do not provide information about the operations performed on the different dam elements during the simulation: they only return how the dam level and the water released may evolve to ensure its safety. These issues have inspired us to develop a tool that complements existing ones. Our approach uses model checking techniques to synthesize operations over dams that ensure dam safety. Additionally, in our proposal the dam is modeled as a hybrid system that takes into account both the continuous evolution of some magnitudes such as dam level, and the discrete behavior of other elements such as the controls of outflow element openings. We have focused on implementing the tool for a particular dam (located in the south of Spain), but the results are obviously applicable to other similar ones.

This work has been partially supported by grants P07-TIC3131 (Andalusia) and TIN-2008-05932 (Spain)

Our previous work focused on developing the model of the dam and its elements using PROMELA as the specification language. We also carried out some tests to see the efficiency of the analysis with SPIN [1]. Compared with [2], the new functionalities added to the tool are the definition of properties by means of timed automata, which makes it easier to define timed properties compared with traditional LTL translation implemented in SPIN, the graphical interface that allows non-expert users to specify these properties and to synthesize operations, and the inclusion of pre-defined outflow policies based on dam operators' experience.

II. Dam Model Definition

We have built a hybrid model of the dam. It is composed of different outflow elements, such as spillways, intermediate outflows and low level outflows. These outflow elements release water following difference equations that depend on dam level and the degree of opening of each element. In the model, it is possible to differentiate continuous variables, such as the dam level or the released water, as well as discrete variables, such as the degree of opening of the outflow elements, which change their position in discrete steps when some external event happens.

The hybrid model includes additional elements apart from the dam. To synthesize operations we have constructed the so-called *user model* that implements a number of interactions with the dam by generating events that open or close the outflow elements. It can be implemented following different outflow policies to produce realistic and useful operations over the dam. At this point we have implemented three different outflow policies with the support of national dam experts. It is worth noting that to synthesize operations, some *non-deterministic* behavior is required in the user model, which may be easily represented in PROMELA.

The modeling and analysis of hybrid systems with PROMELA and SPIN entails solving some important issues, such as the inclusion of a time model and the management of continuous variables. At this stage, we implemented a discrete time model based on the definition of *timer variables*

and a *synchronization process* that is included in the hybrid model. Moreover, to manage continuous variables, we used the C code extension of PROMELA that allows defining real value variables. More information about the time model and the management of continuous variables can be found in our previous work [2].

III. Synthesis of Dam Operations

As mentioned above, our objective is to develop a tool that provides different alternatives to manage a dam in flow episodes. To this end, given a particular scenario, we synthesize different sequences of operations that can be performed to preserve dam safety. This task involves defining some safety properties and analyzing them on the hybrid model, in this case using SPIN. In [2], the properties were expressed by means of LTL formulas. SPIN provides an automatic parser from LTL to a special PROMELA process called *never claim*. However, since LTL is not appropriate to describe properties that refer to precise time instants, we have extended the manner in which properties are defined in SPIN by using timed automata.

To express a valid property with a timed automata, we have to define, as minimum, an initial state and a goal state that is only reached when the property is satisfied. Of course, we can define more intermediate states to monitor different conditions. In addition, guards and invariants have to be defined to control that the goal state is only reached when the dam situation is safe. Figure 1 shows an example of a property expressed as a timed automaton. This automaton will reach the goal state if the dam level is in range `[v1,v2]` when time is in range `[10,100]`.

The translation of timed automata to Büchi automata is almost direct. Discrete locations are translated into *if* blocks. We have labeled both the discrete locations and the transitions to clarify the model. Guard conditions are translated into branches in the *if* blocks with a boolean expression and a final *goto* statement to jump to the target location. Invariant conditions are translated into branches of *if* blocks with boolean expressions and a final *goto* statement to jump to the source location, in order to implement a loop. To improve the performance of the analysis, we have also included some extra conditions in the guard branches that only force the jump when the synchronization process was the last process executed.

Figure 1 shows the timed automaton and the Büchi automaton (SPIN *never claim* process) for the property previously mentioned. This property checks if the dam level is in range `[v1,v2]` when time is in range `[10,100]`. In addition, in our application, if one model execution is found that satisfies the property, we are interested in the dam operations that have been carried out during this execution. That is, we need to have the tool return these operations as a counterexample. However, in SPIN, this is only possible if the analysis of the property fails. This is why the translation

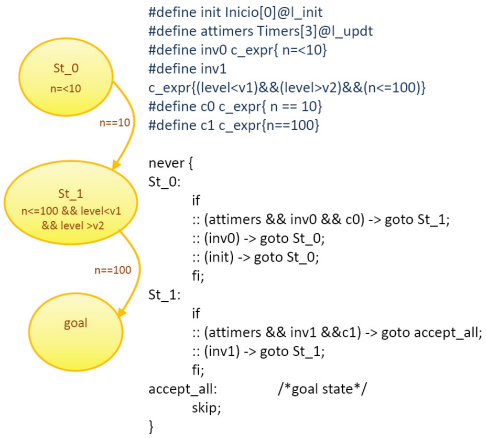


Figure 1. Property definition as a timed automaton

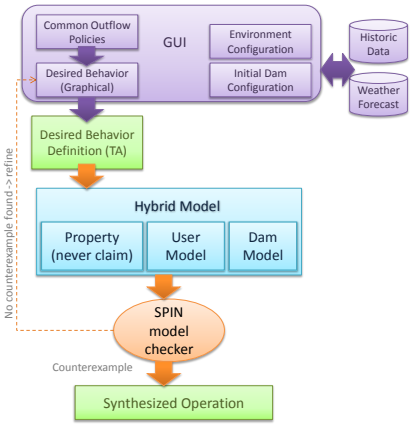


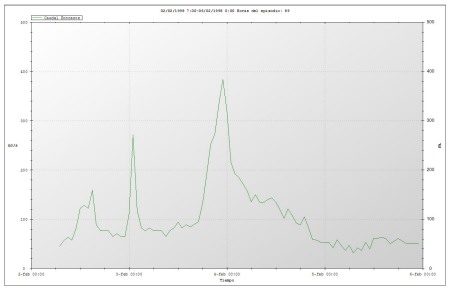
Figure 2. Tool Architecture

from timed automata to Büchi automata defines the final state as unsafe (erroneous) in the *never claim* process. This is achieved declaring the original goal state as an *acceptance state* in the resulting Büchi automata. During the analysis, if SPIN reaches the acceptance state, the analysis ends, and it returns the sequence of states leading to this *error* state as a counterexample. This sequence corresponds to the scheduling of operations that preserve the original property.

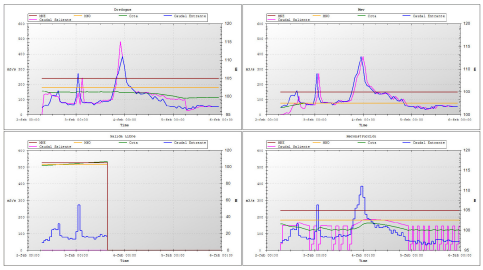
Also, since time evolution is increasing and monotonic, we can discard those execution paths where time does not fulfill the time conditions. Additionally, the way of implementing the synchronization process assures that state transitions are only possible after the synchronization process has been executed. The remaining processes cannot transit, even though the transition condition holds, until the synchronization process is executed.

IV. Automatic Decision Support Tool

We have completed the approach with a user-friendly interface that allows the final user (dam operators) to easily

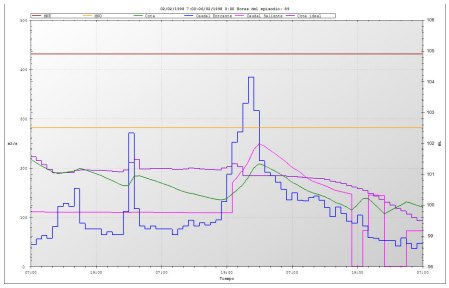


(a)

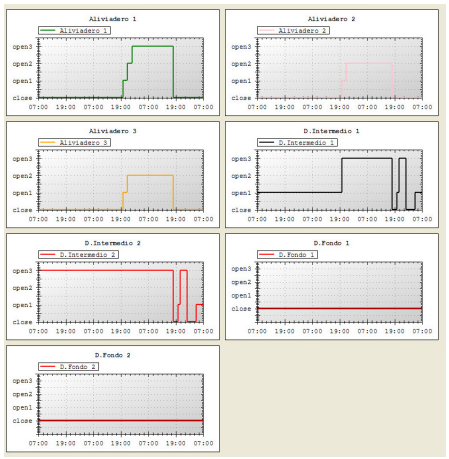


(b)

Figure 3. Tool execution



(a)



(b)

Figure 4. Analysis results

introduce and review all the input/output data. Figure 2 shows the tool's global architecture. It includes the engine to perform property synthesis. In addition, it provides mechanisms to obtain the environmental inflow from different sources: historic data stored in data bases, weather forecasts, or data coming from a hydrological simulation tool. To make it easy to define the desired behavior for the dam (the property to be analyzed), the tool implements different outflow policies.

Figures 3 and 4 illustrate how the tool can be used. The methodology can be summarized as follows: first, the user defines the period of time he/she wants to analyze and selects the environmental inflow. In this example (as Figure 3(a) shows), we have selected a historic flood episode of 72 hours. Then, these input data and the initial state of the outflow elements allow us to perform several analyses. On the one hand, we can analyze the dam model using the three different outflow policies implemented and, on the other, it also possible to carry out an additional analysis where no policy is applied, meaning that the initial state of each outflow element is preserved. Figure 3(b) shows the theoretical curves of the dam level and the water released after applying the procedures provided by each policy (including the case where no policy is applied.) It also shows the original inflow and the maximum and normal operational levels. It is worth noting that the first phase does not ensure that the dam evolution proposed by the policies can be achieved in a realistic manner.

To perform the synthesis of operations, the next step is to select one of the policies and one particular magnitude (the dam level or the water released) as the desired behavior. In the example, we have selected the evolution of dam level in the first policy (the top left graph in the Figure 3(b)). For this analysis, it is also necessary to set an error margin, that is, the variation with respect to the desired behavior that can be assumed. In this case, we have considered an upper margin of +0.5m. and a lower margin of -1.1m. with respect to the desired dam level evolution. These constraints correspond to the properties that the model evolution should satisfy. They are translated into a timed automaton and later to a *never claim* process. Considering this configuration, the user may perform a second analysis. If the tool can find a solution, it returns the counterexample containing the real evolution of the dam level and the water released, and also the set of operations to be performed in each outflow element (as shown in Figures 4(a) and 4(b)). If no solution is found, the user can launch a new analysis (maybe modifying the error margins) with looser limits. It is also possible to graphically modify the desired behavior and launch a new analysis.

V. Conclusions

In this short paper, we have briefly described how to use model checking techniques for dam management. As a novel contribution, a particular dam has been modeled as a hybrid

system using PROMELA as the specification language and SPIN as the tool to perform the analysis of safety properties. Since SPIN is a tool for discrete system analysis that uses LTL to express properties, we have extended the manner in which properties are defined to include time. To this end, we have proposed the use of timed automata and their translation into *never claim* processes.

Finally, all these features have been integrated into a tool for dam management. This tool provides useful information to dam operators about how to manage the different outflow elements to achieve a desired behavior. The tool also permits obtaining inflow information from external sources. In addition, it implements different outflow policies to provide operators with different alternatives for the dam evolution and, finally, it allows the synthesis of the operations over the time needed to reach a safe final state for the dam.

Future work is oriented to making an extension of PROMELA and SPIN to improve the performance of hybrid system analysis. The current prototype is fully operative for flood periods up to 72 hours. However, if we analyze longer periods it is possible to find state space explosion problems. For that reason, we would like to include time abstraction techniques so that time does not have so much influence in the analysis.

ACKNOWLEDGMENT

The authors would like to thank Agustín Merchán, Spanish expert in dams, for his help.

REFERENCES

[1] G. J. Holzmann, *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[2] M. del Mar Gallardo, P. Merino, L. Panizo, and A. Linares, "Developing a decision support tool for dam management with spin," in *14th International Workshop, FMICS 2009, Eindhoven*, ser. LNCS, Nov. 2009, pp. 210–212.

Dam Management based on Model Checking Tecniques

María-del-Mar Gallardo[1], Pedro Merino[1], Laura Panizo[1], David Sanán and Antonio Linares[2]
[1]Dept. Lenguajes y Ciencias de la Computación. University of Málaga.
{gallardo, pedro, laurapanizo,sanan}@lcc.uma.es
[2] BEFESA AGUA, SAU
Antonio.linares@befesa.abengoa.com

1. Motivation

- Dams are critical systems traditionally managed by human operators. Extreme and unexpected situations can lead to human errors.
- In recent years, tools based on numerical models and simulations have emerged to help dam operators.
- We propose the use of hybrid models and model checking, to analyze these critical systems. Our main objective is to synthesize operations that ensures the safety of the dam using properties. Table 1 shows a comparison between traditional methods and our approach.
- We use PROMELA formal language and SPIN to perform the analysis. To synthesize the operation we define some properties as timed automata.

	Traditional Dam Management Method	Proposed Dam Management Method
Based on	Numerical Models and Simulations	Formal Models and Exhaustive Search
Time to evaluate one set of operations	One simulation (1-2 h.)	One execution with deterministic user model (1-5 min.)
Time to Synthesize operations	Several simulations	One execution with non-deterministic user model (5-15 min.)
Synthesis result	Total outflow	Total outflow + operation set
Accuracy continuous part	High	Medium (can be adapted to memory and time req.)
Accuracy discrete part	Low	High

Table 1. Comparison between traditional dam management and our proposal

2. Dam Model Definition

- The dam is modeled as a hybrid system. Continuous magnitudes, such as water level, evolve continuously following difference equations.
- External events can modify the state of dam elements and forces a change in the equation. Figure 1 shows the elements of the hybrid model:
  - Dam and outflow elements: spillways, intermediate outflows and low level outflows. All of them have continuous variables with time dependency (water level, water released) and discrete variables (opening degree of the outflow elements)
  - User model: It generates external events that modify the state of dam elements. There are different implementations to represent different outflow policies. In addition, user model presents some non-deterministic behavior.
- Modeling a hybrid system with PROMELA implies:
  - The development of a time/synchronization model to include the time dependency of continuous variables -> Discrete time model (Figure 2).
  - Description of the continuous behavior -> PROMELA C code extension.

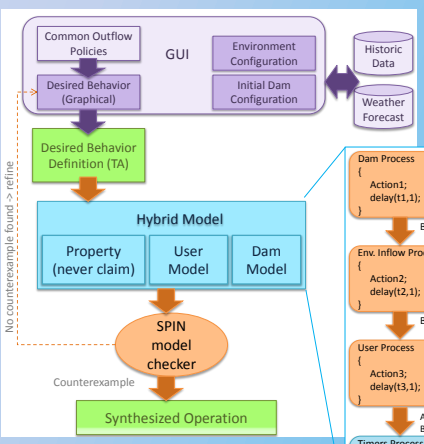


Figure 1. Tool Architecture

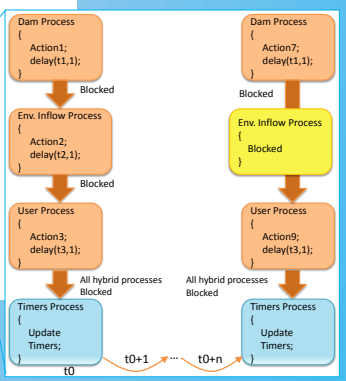


Figure 2. Time/Synchronization model

3. Synthesis of Dam Operations

- The synthesis of operations is based on the verification of properties with the SPIN model checker.
- The desired behavior is defined graphically as curves. Internally, that curves are represented with timed automata (TA). Then, TA are translated to a *never claim* (Büchi automaton) used by SPIN for safety property analysis.
- Figure 4 shows an example of TA to never claim translation. It includes optimizations for timers variables, to avoid checking useless states.
- In TA, to satisfy a desired behavior the goal state has to be reached. In the never claim, the goal state is translate into an acceptance state (erroneous state). If SPIN reaches it, the analysis ends and a counterexample is returned and used to synthesize the correct operations over the dam.

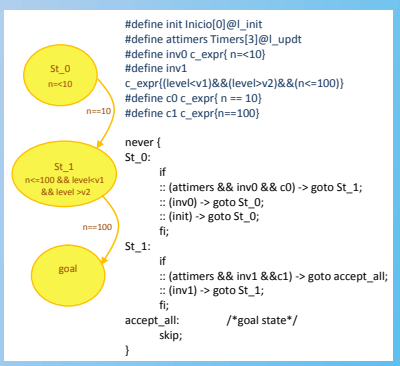


Figure 3. Translation form TA to *never claim*

4. Automatic Decision Support Tool

- We have completed the approach with a user-friendly interface. Figure 1 shows the global architecture of the new tool.
- This tool allow the analysis of flow periods up to 72 hours. In this interval, the time to analysis is lower than 20 minutes and the memory needed is always under 1GB.
- Figure 4 shows an example of use. Given the date and duration of the flow period
  - The tool apply different classical outflow policies, returning theoretical behaviors. One of them is selected.
  - The tool works out the operations to achieve an approximation of the selected behavior.

5. Conclusions

- The use of hybrid formal methods in dam management is producing very promising results.
- We have use PROMELA to model the system and SPIN synthesize operations. We have implemented a discrete time model and used embedded C code extension to manage continuous variables in SPIN.
- We propose an automatic translation from timed automata to never claim to define properties/desired behavior reducing the number of checked states.
- All these features have been integrated in a decision support tool for dam management.

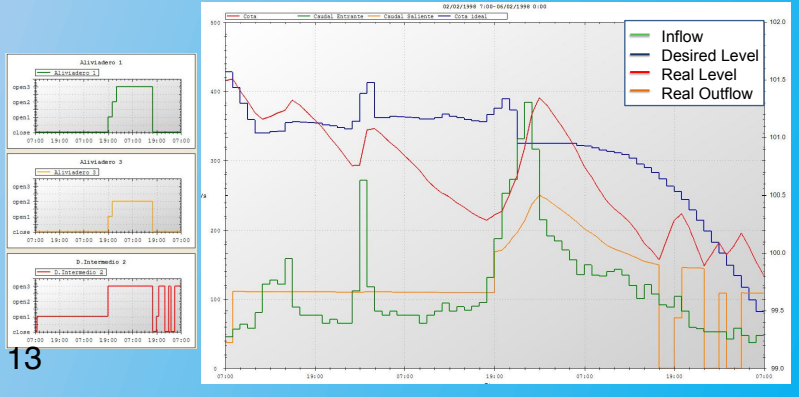


Figure 4. Tool use case

Partially supported by Befesa Agua and by grants P07-TIC3131 (Andalusia) and TIN2008-05932 (MEC and FEDER)

Acknowledgments:
*Authors would like to thank Agustín Merchán, Spanish expert in dams,*for his help.

EB2C : A Tool for Event-B to C Conversion Support

Dominique Méry
Université Henri Poincaré Nancy 1
LORIA, BP 239, 54506
Vandoeuvre lès Nancy, France
Email: mery@loria.fr

Neeraj Kumar Singh
Université Henri Poincaré Nancy 1
LORIA, BP 239, 54506
Vandoeuvre lès Nancy, France
Email: singhnne@loria.fr

***Abstract*—To use of formal model effectively in formal method based development process, it is highly desirable that the formal specification be converted to C code, a de facto standard in many industrial application domains, such as medical, avionics and automotive control. In this paper we present the design methodology of a tool that translates an Event-B formal specification to equivalent C code with proper correctness assurance.**

***Keywords*-C, Formal model, Event-B, Proof-based refinement**

I. Introduction

Recently Medical, Avionics and Automotive industries are leaning more and more towards formal method-based development of safety-critical software. In embedded system community, formal method-based development implies verification and validation, and generated proof obligation ensure proof of correctness of a system model. The auto code generators associated with formal development tools can generate software codes from the formal specifications, thus enabling model developer to generate source code automatically without knowing the target language syntax. Proof-based development methods [1] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the (closed) system under development. Details and design choices are introduced in an incremental way. The correctness between two levels is ensured by refinement proofs. It also maintains very well refinement techniques that can transform an abstract and non-deterministic specification into a concrete, deterministic system model, in several stages. When implementations are aimed, refinement leads to a last level which describes, in some way, the expected behavior. Refinement-based model development is very popular in many industrial application domains. It is considered as a de facto standard in very complex systems such as in medical, avionics and automotive domains. Medical, Avionics and Automotive industries are already in possession of large formal specifications developed in formal language, which are not able to generate source code in C language [2] automatically. Realizing the benefit of adhering to automatic source code generation in their formal development process, these industries are now interested in embracing automatic source code generation approaches in their new projects; thereby having source codes for all of their old formal specifications for the ease of reuse and maintenance.

In this paper, we describe an approach to build a tool which will be useful to translate Event-B [1] formal specification model to an equivalent C code function. The implementation of translation is described in the form of a *toolchain*. We also provide a mechanism to ensure the correctness of such a translation. From Classic-B [3] notation to C language translation tool has been developed by D. Bert, et al. [4], in which models are restated in an intermediate language "B0", and then converted to finally in C language. A pioneering work for automatically translate subset of Event-B formal notation of MIDAS [5] specification in C language is proposed by S. Wright [5]. The shortcoming of the work is that the tool shows only for simple translation of formal concrete machines. Moreover, in Wright's work, handling of constants, axioms, enumerated sets and functions are not covered in translation process. In our work, we provide complete translation for them. For large generated source code, optimization using events scheduling and translation correctness assurance are of prime importance. We show how, event scheduling is resolved by refinement approach. Proper correctness assurance of generated C code from Event-B modeling language is verified using the code verification tool.

II. Description of the Tool

The translation process consists in transforming the concrete part of an Event-B project into a semantically equivalent text written in C programming language. We propose an architecture for the Event-B translator. Figure-1 depicts the overall architecture of the tool. The tool is called EB2C. This tool has mainly four components: Pre-processing, Event-B to C translator, code optimization and code verification. The input of the translator tool is a Rodin project [6] files containing formal specification in Event-B modeling language. To generate C code for an Event-B model we use Eclipse development framework for developing a plug-in in the Java language. The Pre-processing takes an Event-B project and introduce C context file to provide deterministic range for all kind of data types and makes an Event-B model

Event-B type	Formal Range	C type
tl_int16	$-2^{15}..2^{15}-1$	int
tl_uint16	$0..2^{16}-1$	unsigned int
tl_int32	$-2^{31}..2^{31}-1$	long int
tl_uint32	$0..2^{32}-1$	unsigned long int

Table I
INTEGER BOUNDED DATA TYPE DECLARATION IN CONTEXT FILE

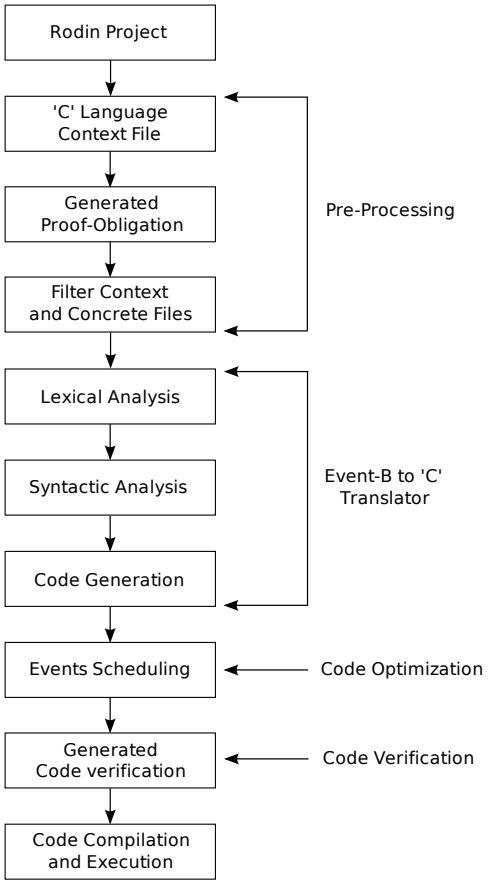


Figure 1. Architecture of Translation Tool

deterministic. Table-I shows bounded integer data types of C context.

The next two sub-steps of the Pre-processing is to prove all generated proof obligations and filter context files and concrete machines files from selected Rodin project [6]. An Event-B to C translator takes set of context and concrete filtered files. In this translation process our aim is to generate a C code file for an Event-B concrete file using Event-B grammar through syntax-directed translation. The translator generates separate functions for these Event-B events. We consider Event-B formal notations available at [1]. We capture Event-B grammar in a Abstract Syntax Tree (AST). For the production rules of the Event-B grammar we designed the appropriate algorithm with appropriate parameters to generate desired C codes in the targeted file with ".c" extension during the parsing process. The translator successfully translates an Event-B model with the following components: Constants, Enumerated sets, Functions, Variables, Arrays, Parameters, Events, Guards and Actions with arithmetic and logical operations. At present the translator cannot translate a sets, sets operations and relation over sets. Event-B has a rich set of formal model elements which cover formal modeling as abstraction, which is not applicable for code translation. We have closely examined the Event-B grammar, and the observational equivalence between Event-B and C programming language types is done as follows:

Event-B types	C Language types
Enumerated sets	Enumerated types
Basic integer sets	Predefined integer types
Event-B array types	C array types
Function	C function structure

Table II shows a set of Event-B syntax to an equivalent C programming language. This table shows a list of supported Event-B syntax. All constants defined in a model's context must be replaced with their literal values. This translation tool supports conditional, arithmetic and logical expression of formal model and translates into equivalent C language code.

An Event-B enumerated sets is equivalent to the C programming language enumerated types. It is very easy to translate into C programming language equivalent form due to equivalent semantical structure.

The links between Event-B and C programming language for integer values have been considered as crucial for the efficiency of the generated code and for the correctness of the translation. So, the solution is provided in first level of Pre-processing phase by introducing C programming language context and it is able to interface very tightly with Event-B integer types and C programming language integer types.

The links between Event-B arrays and C programming language arrays are not straightforward. In Event-B, arrays correspond to total functions whereas in C programming language, they correspond to a contiguous zone of memory (coded as the beginning address of the array and its size). However, it is easy to do a semantical correspondence between an array element *arr(i)* in Event-B and the value at the location *arr[i]* in C programming language.

The links between Event-B functions and C programming language functions are also very ambiguous. Translation tool only supports total functions of Event-B into equivalent corresponding C programming language functions. However, it is easy to do a semantical correspondence between a function passing parameters in a C programming language is equivalent to the elements of left side of total functions symbol ($\rightarrow$) and output of the C programming language function correspond to the right hand side of the

Event-B	C Language	Comment
n..m	int	Interger type
x ∈ Y	Y x;	Scaler declaration
x ∈ tl_int16	int x;	'C' Context declaration
x ∈ n..m → Y	Y x [m+1];	Array declaration
x :∈ Y	/* No Action */	Indeterminate initialization
x : | Y	/* No Action */	Indeterminate initialization
x = y	if(x==y) {	Conditional
x ≠ y	if(x!=y) {	Conditional
x < y	if(x<y) {	Conditional
x ≤ y	if(x<=y) {	Conditional
x > y	if(x>y) {	Conditional
x ≥ y	if(x>=y) {	Conditional
(x>y) ∧ (x≥z)	if ((x>y) && (x>=z) {	Conditional
(x>y) ∨ (x≥z)	if ((x>y) ‖ (x>=z) {	Conditional
x := y + z	x = y + z;	Arithmetic assignment
x := y - z	x = y - z;	Arithmetic assignment
x := y * z	x = y * z;	Arithmetic assignment
x := y ÷ z	x = y / z;	Arithmetic assignment
x := F(y)	x = F(y);	Function assignment
a := F(x↦y)	a = F(x, y);	Function assignment
x := a(y)	x = a(y);	Array assignment
x := y	x = y;	Scalar action
a := a ⩤ {x↦y}	a(x) = y;	Array action
a := a ⩤ {x↦y} ⩤ {i↦j}	a(x)=y; a(i)=j;	Array action
X⇒Y	if(!X ‖ Y){	Logical Implication
X⇔Y	if((!X ‖ Y) && (!Y ‖ X)){	Logical Equivalence
¬x<y	if(!(x<y)){	Logical not
x ∈ ℕ	unsigned long int x	Natural numbers
x ∈ ℤ	signed long int x	Integer numbers
∀	/* No Action */	Quantifier
∃	/* No Action */	Quantifier
Sets	/* No Action */	Sets operations

fun ∈ ℕ × ℕ → ℕ	long int fun(unsigned long int $arg1$, unsigned long int $arg2$)
	{
	//TODO: Add your Code
	return;
	}
		Function Definition

Table II
EVENT-B TO C TRANSLATION SYNTAX

total functions symbol (→) in Event-B. So, this step of function translation generates the function structure into C programming language. If body of the function is defined in Event-B as in form of predicate then translation tool translate equivalent predicate into function body of the C programming language.

An Event-B model is a collection of interdependent events. The translator takes each of these events one by one and converts them to an equivalent C function. So we have separate C functions for each events. This C code file contains appropriately generated constants, local and global variables, arrays, functions, and events which are generated from Event-B models using lexical and syntactic analysis.

Translation tool provides a recursive process to generate source code for each event of the Event-B specification into C programming language. Translation tool always checks for "null" event (i.e. guard of false condition), never generates the source code for that event and inserts suitable comment into the source code for traceability purpose. This automatic reduction is performed to avoid generation of unreachable run-time code.

In Event-B specification, there are two kind of variables: global variables and local variables. Global variables are derived directly from VARIABLES statements of the concrete machine and all these variables have global scope. Local variables are derived from the ANY statement of the particular event, and are entirely local to the corresponding function. Therefore no parameters are passed to C programming language function. The function returns a boolean value, signaling whether the event has been triggered to its calling environment at run-time. After generation of the function header, all local variable declarations are inserted at the beginning of the function, giving them scope across the whole function.

In Event-B guard handling is very ambiguous due to different meaning, such as local variables type definition, the assignment of a value to a local variable, condition statements using negation (¬), conjunction (∧), disjunction (∨), implication (⇒) and equivalence (⇔) operators. Therefore, for handling so much complex situation, we introduce a recursive algorithm for parsing complex guards and separate each element of guard for translation purpose. Thus each

guard must be automatically analyzed to resolve this ambiguity from context information. For example an implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$) operator, we rewrite the predicate in equivalent form using conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$) operators, the equals relation may signify an assignment or equality comparison, and the precise meaning (and hence the resulting translation) deduced from the type and scope of its operands. A further ambiguity that must be resolved is the meaning of a functional-image relation, which may be used to model a data array or an external function. Once the guards of the event have been classified, those conferring local variable type information are used for variable declarations in the function, and remaining guards are used to generate local assignment and conditional statements. Local variable type information is derived in a similar fashion as the global variables from the guard information instead of using INVARIANT.

All condition guards are placing in the function as nested statements, using directly translated conditional and local variables declared within nested scope ranges. After insertion of all conditional guards, we provide translation for remaining local assignments with a comment. The meaning of functional-image statements within the model is automatically resolved to an array if the mapping is a global variable, otherwise a call to an uninterpreted function is inserted.

The next sub-stage of event translation is the actions translation. In Event-B, actions are triggered in a concurrent manner and state that all state modification in the actions is only valid in the entire event post-condition. Therefore, dependency checks must be performed to ensure that any state variable used as an action assignee has not already been modified to its post-condition prior to use. A similar way of parsing is applied on Event-B action statement as a guard statement. An action translation supports assignments to scalar variables, override statements acting on array-type variables and arithmetic complex expression. The translation tool translates all Event-B actions into equivalent C programming language source code.

The code optimization is used to optimize the events scheduling for calling functions using refinement approach. An incremental refinement-based structure of events within an Event-B model exploit to recursively generate nesting calling functions corresponding to the abstract events. An abstract level guards are forming a group of concrete events. An event group is inserted for execution in place of multiple events, improving run-time performance.

We propose two techniques to trigger all translated events. First is a calling function *"Iterate"* implements a continuous iteration of translated C programming language functions of the Event-B model, in the same order, defined by their position in the Event-B model. Second technique is to optimize the calling order of the events. From the event scheduling, we have got the code structure that help to make a group of all concrete events. Each group of events are triggered by main *"Iterate"* function.

The code verification is very important to verify correctness of translated code. We use a methodology called code verification to validate a C code with respect to the formal specification from which it has been generated. Main objective to use code verification to test the C program with proper coverage criteria and to ensure its correctness. In this way, C codes for all the Event-B formal models are successfully generated.

III. Conclusion

Generating C code from Event-B formal specification is of prime importance to medical, avionics and automotive industries. This importance arises from lots of advantages that can be gained using formal methods based development process in the design, development and maintenance of embedded software. The medical, avionics and automotive industries are willing to convert all their formal specifications to equivalent C codes, and then leverage formal methods based development process for further work. The approach we discussed in this paper will be of great help towards this goal. We have implemented the translator covering a large subset of Event-B modeling language. We have also completely automated the test of translated code generation process for correctness assurance. The translator tool has been tested on a number of Event-B formal models of a cardiac pacemaker system with encouraging results. In near future, we have planed to extend this translation tool to handle *Sets*, *Relation* and other kind target programming language such as *ladder logic*, so that this translation tool can be used by all industrial areas, whereas formal verification and validation are primary techniques to develop a system.

Acknowledgement Work of Dominique Méry and Neeraj Kumar Singh is supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche. Neeraj Kumar Singh is supported by grant awarded by the Ministry of University and Research.

References

[1] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] B. W. Kernighan and D. Ritchie, *C Programming Language*. Prentice Hall, 1988, iSBN-100131103628.

[3] J.-R. Abrial, *The B-book: assigning programs to meanings*. New York, NY, USA: Cambridge University Press, 1996.

[4] D. Bert, S. Boulmé, M.-L. Potet, A. Requet, and L. Voisin, *Adaptable Translator of B Specifications to Embedded C Programs*. FME 2003, Springer, 2003, pp. 94–113.

[5] S. Wright, "Automatic generation of c from event-b," in *Workshop on Integration of Model-based Formal Methods and Tools*, February 2009. [Online]. Available: http://www.cs.bris.ac.uk/Publications/Papers/2000990.pdf

[6] Project RODIN, "Rigorous open development environment for complex systems," http://rodin-b-sharp.sourceforge.net/, 2004.

[7] Dominique Méry and Neeraj Kumar Singh, "Functional behavior of a cardiac pacing system (in Press)," *International Journal of Discrete Event Control Systems*, vol. 1, 2010.

IV. Appendix

We use our developed translation tool to generate source code in 'C' language from Event-B modeling language of a cardiac pacemaker specification. The cardiac pacemaker system is a medical device which uses electrical impulses, delivered by electrodes contacting the heart muscles, to regulate the beating of the heart. The primary purpose of a pacemaker is to maintain an adequate heart rate, either because the heart's native pacemaker is not fast enough, or there is a block in the heart's electrical conduction system [7]. This case study completes a formal development of the cardiac pacemaker using Event-B. We use our developed translation tool to generate source code in 'C' language from cardiac pacemaker Event-B specification. The generated code is 90% equivalent to the handwritten code. Remaining 10% coding is additional coding style which provided by the developer in the pacemaker system using some other language. So, we believe that our translation tool is correct. An example of an Event-B event and its 'C' translation is given as follows:

Actuator ON and OFF events of Event-B

EVENT Actuator_ON_V
WHEN
grd1 : $PM\_Actuator\_V = OFF$
grd2 : $(sp = Pace\_Int)$
$\vee$
$(sp < Pace\_Int \wedge$
$AV\_Count > V\_Blank \wedge$
$AV\_Count \geq FixedAV)$
grd3 : $sp \geq VRP \wedge sp \geq PVARP$
THEN
act1 : $PM\_Actuator\_V := ON$
act2 : $last\_sp := sp$
END

EVENT Actuator_OFF_V
WHEN
grd1 : $PM\_Actuator\_V = ON$
grd2 : $AV\_Count\_STATE = TRUE$
grd3 : $PM\_Sensor\_A = OFF$
grd4 : $PM\_Actuator\_A = OFF$
grd5 : $sp \geq VRP \wedge sp \geq PVARP \wedge sp \geq URI$
grd6 : $(sp = Pace\_Int)$
$\vee$
$(sp < Pace\_Int \wedge$
$AV\_Count > V\_Blank \wedge$
$AV\_Count \geq FixedAV)$
THEN
act1 : $PM\_Actuator\_V := OFF$
act2 : $AV\_Count\_STATE := FALSE$
act3 : $AV\_Count := 0$
act4 : $PM\_Sensor\_V := OFF$
act5 : $sp := 1$
act6 : $Thr\_A := 0$
act7 : $Thr\_V := 0$
act8 : $Thr\_A\_State := FALSE$
act9 : $Thr\_V\_State := FALSE$
END

Translated C code

```
BOOL Actuator_ON_V(void)
{
    /* Guards No. 1 */
    if(PM_Actuator_V == OFF){

    /* Guards No. 2*/
    if((sp == Pace_Int) ||
    ((sp < Pace_Int) &&
    (AV_Count > V_Blank) &&
    (AV_Count >= FixedAV))) {

    /* Guards No. 3 */
    if((sp >= VRP) &&
    (sp >= PVARP) {

    /* Actions */
    PM_Actuator_V = ON;
    last_sp = sp;

    return TRUE;
    }}}
    return FALSE;
}
```

```
BOOL Actuator_OFF_V(void)
{
    /* Guards No. 1 */
    if(PM_Actuator_V == ON){

    /* Guards No. 2*/
    if(AV_Count_STATE == TRUE){

    /* Guards No. 3*/
    if(PM_Sensor_A == OFF){

    /* Guards No. 4*/
    if(PM_Actuator_A == OFF){

    /* Guards No. 5*/
    if((sp >= VRP)&&(sp > PVARP)&&(sp >= URI)){

    /* Guards No. 6*/
    if((sp == Pace_Int) ||
    ((sp < Pace_Int) &&
    (AV_Count > V_Blank) &&
    (AV_Count >= FixedAV))) {

    /* Actions */
    PM_Actuator_V = OFF;
    AV_Count_STATE = FALSE;
    AV_Count = 0;
    PM_Sensor_V = OFF;
    sp = 1;
    Thr_A = 0;
    Thr_V = 0;
    Thr_V_State = FALSE;
    Thr_A_State = FALSE;

    return TRUE;
    }}}}}}
    return FALSE;
}
```

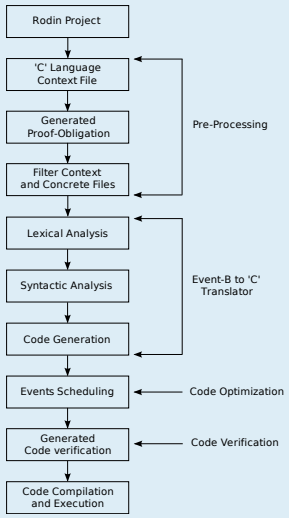
EB2C : A Tool for Event-B to C Conversion Support

Dominique Méry and Neeraj Kumar Singh
LORIA & Université Henri Poincaré Nancy 1,
BP 239, 54506, Nancy, France

Abstract

To use of formal model effectively in formal method based development process, it is highly desirable that the formal specification be converted to C code, a de facto standard in many industrial application domains, such as medical, avionics and automotive control. Here, we present the design methodology of a tool that translates an Event-B formal specification to equivalent C code with proper correctness assurance.

Architecture of Translation Tool



Conclusion

Generating C code from Event-B formal specification is of prime importance to medical, avionics and automotive industries. This importance arises from lots of advantages that can be gained using formal methods based development process in the design, development and maintenance of embedded software. The medical, avionics and automotive industries are willing to convert all their formal specifications to equivalent C codes, and then leverage formal methods based development process for further work. The approach we discussed in this paper will be of great help towards this goal. We have implemented the translator covering a large subset of Event-B modeling language. We have also completely automated the test of translated code generation process for correctness assurance. The translator tool has been tested on a number of Event-B formal models for encouraging results.

References

[1] J.-R. Abrial. Modeling in Event-B: System and Software Engineering, In *Cambridge University Press, 2010.*

[2] J.-R. Abrial. Modeling in Event-B: System and Software Engineering, In *Cambridge University Press, 2010.*

[3] J.-R. Abrial. The B-book: assigning programs to meanings, In *New York, NY, USA: Cambridge University Press, 1996.*

[4] D. Bert, S. Boulmé, M.-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C, In *Programs. FME 2003, Springer, 2003, pp. 94-113.*

[5] S. Wright. Automatic generation of c from event-b, In *in Workshop on Integration of Model-based Formal Methods and Tools, February 2009.*

[6] Project RODIN. Rigorous open development environment for complex systems, In *http://rodin-b-sharp.sourceforge.net/, 2004.*

Description of the Tool

An architecture of translation tool depicts various phases of translation tool. The tool is called EB2C. This tool has mainly four components: Pre-processing, Event-B to C translator, code optimization and code verification. The input of the translator tool is a Rodin project [6] files containing formal specification in Event-B modeling language. The Pre-processing takes an Event-B project and introduce C context file to provide deterministic range for all kind of data types and makes an Event-B model deterministic. Bounded integer data types of C context are given in following table:

Event-B type	Formal Range	C type
tl_int16	$-2^{15}..2^{15}-1$	int
tl_uint16	$0..2^{16}-1$	unsigned int
tl_int32	$-2^{31}..2^{31}-1$	long int
tl_uint32	$0..2^{32}-1$	unsigned long int

The next two sub-steps of the Pre-processing is to prove all generated proof obligations and filter context files and concrete machines files from selected Rodin project [6]. An Event-B to C translator takes set of context and concrete filtered files. In this translation process our aim is to generate a C code file for an Event-B concrete file using Event-B grammar through syntax-directed translation. The translator successfully translates an Event-B model with the following components: Constants , Enumerated sets, Functions, Variables, Arrays, Parameters, Events, Guards and Actions with arithmetic and logical operations. We have closely examined the Event-B grammar, and the observational equivalence between Event-B and C programming language types is done as follows:

Event-B types	C Language types
Enumerated sets	Enumerated types
Basic integer sets	Predefined integer types
Event-B array types	C array types
Function	C function structure

The code optimization is used to optimize the events scheduling for calling functions using refinement approach. An incremental refinement-based structure of events within an Event-B model exploit to recursively generate nesting calling functions corresponding to the abstract events. An abstract level guards are forming a group of concrete events. An event group is inserted for execution in place of multiple events, improving run-time performance.

We propose two techniques to trigger all translated events. First is a calling function *"Iterate"* implements a continuous iteration of translated C code functions of the Event-B model, in the same order, defined by their position in the Event-B model. Second technique is to optimize the calling order of the events.

The code verification is very important to verify correctness of translated code. We use a methodology called code verification to validate a C code with respect to the formal specification from which it has been generated. Main objective to use code verification to test the C program with proper coverage criteria and to ensure its correctness.

Event-B to C Translation

We use our developed translation tool to generate source code in 'C' language from Event-B modeling language of a cardiac pacemaker specification. A small sample of equivalent translated code is given as follows:

EVENT Actuator_ON_V
WHEN
grd1 : $PM\_Actuator\_V = OFF$
grd2 : $(sp = Pace\_Int)$
$\vee$
$(sp < Pace\_Int \wedge$
$AV\_Count > V\_Blank \wedge$
$AV\_Count \geq FixedAV)$
grd3 : $sp \geq VRP \wedge sp \geq PVARP$
THEN
act1 : $PM\_Actuator\_V := ON$
act2 : $last\_sp := sp$
END

An event of Event-B

```
BOOL Actuator_ON_V(void)
{
    /* Guards No. 1 */
    if(PM_Actuator_V == OFF){

    /* Guards No. 2*/
    if((sp == Pace_Int) ||
    ((sp < Pace_Int) &&
    (AV_Count > V_Blank) &&
    (AV_Count >= FixedAV))) {

    /* Guards No. 3 */
    if((sp >= VRP) &&
    (sp >= PVARP) {

    /* Actions */
    PM_Actuator_V = ON;
    last_sp = sp;
    return TRUE;
    }}}
    return FALSE;
}
```

Translated C code

Work in Progress

We are concentrating to extend this translation tool to handle *Sets*, *Relation* and other kind of target programming language such as *ladder logic*, so that this translation tool can be used by all industrial areas, whereas formal verification and validation are primary techniques to develop a system.

Funding

This work is supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche and Ministry of University and Research.

A tool to verify, slice and animate annotated components: GamaSlicer

Daniela da Cruz, Pedro Rangel Henriques, Jorge Sousa Pinto
Departamento de Informática / CCTC
Universidade do Minho
Braga, Portugal
{*danieladacruz,prh,jsp*}*@di.uminho.pt*

***Abstract*—In this paper we propose to demonstrate the features of GamaSlicer as an online tool to experiment and animate *verification* and *assertion-based slicing* algorithms on annotated components written in Java/JML.**

***Keywords*-Assertion-based Slicing, Program Verification, Labeled Control Flow Graph**

I. Context and Motivation

An *annotated component* is a piece of code (sequence of statements, like a procedure) augmented with preconditions, postconditions and invariants that specify the contract to which the code should be compliant. Code annotation is being widely accepted as an effective programming approach for the development of correct software (recall, for instance, Java annotated with JML, or C# annotated with Spec#). Moreover, annotated programs can be automatically checked and their correctness verified; This design-by-contract methodology also proved to be useful for safety reuse.

Gama is a project devoted to the *exploration of annotated software components*. Under this project we are doing theoretical research on program verification, on code analysis and slicing techniques and on software visualization. Combining these three main areas, we are looking for effective ways to include in the exploration the following features:

- the verification of the code, to prove that it respects the contract;
- the elimination of code that does not contribute to the validity of the precondition or the postcondition;
- the elimination of code that does not contribute to compute the values used by the caller, taking into account all invocation points;
- the verification that each invocation point preserves the precondition.

The first operation is based on the generation of a set of proof obligations that are submitted to a theorem-prover; for that we use a VCGen algorithm [1]. The other three are based on variants of a specification-based slicing algorithm [2].

In this context, GamaSlicer [3] is a web-based system developed to help us prove the feasibility of our theoretical results and to assess their actual usefulness. By now, GamaSlicer implements the first two operations.

More precisely, it implements a VCGen algorithm and calls the Z3 theorem-prover to check the generated proof obligations. The verification infrastructure consists of a program logic used to assert the partial correctness of individual procedures, from which a VCGen algorithm is derived. VCGen generates, from a program, a set of proof obligations whose validity is sufficient for the program to be correct. The complete background can be found in [4].

On the other hand, it implements an algorithm for *assertion-based slicing* [2] to cut off all the statements, in the body of the procedure, that do not affect neither the precondition or the postcondition. In the case of assertion-based slicing, it is necessary to decorate each statement with a pair of logic assertions, the weakest precondition and the strongest postcondition, that are used to decide the statements to be sliced off. Besides this algorithm, our tool also includes the algorithms published over the past 10 years (*postcondition-based slicing* [5]; *predicate slice* [6]; and *specification-based slicing* [5]).

Slicing algorithms work over the control flow graph (CFG) that expresses all the possible execution paths. Adding the assertion-pairs to the edges it is possible to improve the expressiveness CFG. This new representation of the annotated component is called **labeled control flow graph** (LCFG for short). The visualization of the CFG has been used for a long time to display the structure of a procedure, depicting its dynamic behavior; we argue that it is also an effective way to show the result of slicing a program. We also argue that we can use the assertions labeling the graph edges to animate the verification and slicing processes, aiding to understand both complex algorithms.

Using a LCFG, we animate the slicing process, displaying successive pictures of the system in different states. This sequence of images illustrates the system evolution. Lots of graphical issues were taken into account to make the visual representation full of meaning and ease to understand: *icons, shape, color, size, spatial layout*, etc.

In this paper we propose to demonstrate the features of GamaSlicer (see next section) as a web environment to experiment verification and assertion-based slicing algorithms for code comprehension, maintenance, or even for teaching.

II. GamaSlicer, an overview

After giving an overview of GamaSlicer and describing its architecture, we detail each phase and the way the respective outcome is shown.

GamaSlicer is an online laboratory[1] that helps the users to understand the processes of *verifying* and *slicing* Java programs annotated in JML[2]. Working components can be uploaded from our repository or created by the user.

Inspired on the standard compiler architecture, GamaSlicer is organized as a *pipeline* of the following blocks: (1) a Java/JML front-end (FE) that is a recognizer composed by a parser and an attribute evaluator built automatically by the ANTLR parser generator (from the Java/JML attribute grammar); (2) a Verification Condition Generator; (3) a Proof Obligations Code-Generator; (4) a Theorem-Prover (we call an external one instead of building our own); (5) a Slicer; and (6) a Labeled Control Flow Graph (LCFG) Visualizer and Animator.

The GamaSlicer interface is a web page divided in six main tabs, to distribute the information that becomes available after each working step.

After uploading the file—containing a Java class complemented with a JML specification—the source-code is analyzed by the FE and is transformed into an AST . During this first step an Identifiers Table is also built. In the next step, the VCGen (implemented as a tree-walker evaluator) traverses the AST to generate the verification conditions using a tree-pattern matching strategy; each time a tree matches a pattern, it is transformed by the algorithm and marked as *visited*.

- *Tab 1*: displays the Java/JML source program, and the rules applied along the *verification conditions generation* process; statistical information (program size, annotation complexity, etc.) is also shown at the top of the window.
- *Tab 2*: displays the Abstract Syntax Tree (AST) generated by the FE.
- *Tab 3*: contains the Identifiers Table built during the analysis phase (also at step one).

In a third step, the new AST decorated with verification conditions attached to the nodes is traversed by the Code-Generator (another tree-walker evaluator) to emit the proof obligations. The tool outputs *proof obligations* in SMT-Lib (*Satisfiability Modulo Theories Library*) language. We chose SMT-Lib as it is nowadays the language most widely used by automatic provers, including Z3, Alt-Ergo, and `Yices`. Then, if required, an external Theorem-Prover is called to prove the generated SMT formulae.

- *Tab 4*: contains the generated SMT code; it will also display a table with the verification status of each formula (`sat`, `unsat`, `unknown`) after calling a Theorem-Prover; for each Prover invoked, the time consumed to prove the formulae is also displayed (in this way, the user can compare the efficiency of the provers and draw his or her own conclusions).

In parallel with step three, a fourth step can be performed. Using the AST to derive the LCFG for the Java/JML source-class, the Assertion-based Slicer detects and removes all the statements that are not relevant to the satisfaction of the pre / postcondition.

- *Tab 5*: displays the new program produced by the Slicer; useless statements (those that are cut off) are not actually removed, but shown in strike-out style and in red color.

The fifth step corresponds to the visualization/animation phase. Before the slicing process, the LCFG is shown to exhibit data and control flow inside the class, emphasizing also the weakest precondition and the strongest postcondition pairs that will be used to decide the statements to be sliced off. Then the verification of the implications between assertion-pairs starts, highlighting the edge that corresponds to the pair being tested – the color depends on the logical value returned by the prover. The successive illumination of the various edges produces the animation of the process; the animation is presented in an interactive mode, allowing the user to control the process. At the end the sliced LCFG is shown.

- *Tab 6*: display the LCFG as the visual representation of the annotated component; nodes in the graph that were sliced away are marked in red.

GamaSlicer provides: the possibility to expand and collapse nodes; a *double-side history*, this is, an *image-based history* (displaying the pictures obtained in the previous steps), and a *textual-based history* (displaying the results of every previous verification); the possibility to change the colors according to the user preferences; it also highlights the nodes being animated, making easier to follow the animation. We believer these elements contribute to a richer interactive user experience.

III. An Illustrative Example

To illustrate the several features of GamaSlicer, a set of examples will be prepared in order to show different results. However, to the purpose of this section we will use just one as a running example: the MinMax method. The same program was also the running example in the well-know paper by *Chung et al* in [5] and can be described as follows:

> Given two integers, compute the *absolute minimum* and *absolute maximum*. When the inputs are of the same sign, the *absolute minimum* should have the value of input with the smaller absolute value and the *absolute maximum* should have the value of input with the larger absolute value.

[1] Available at http://gamaepl.di.uminho.pt/gamaslicer.

[2] The standard specification language for Java.

Otherwise, it is meaningless to compare inputs. Accordingly, both of the outputs should be 0.

It is worth to point out that the specification-based algorithm introduced in [5], was neither implemented nor given proofs that it works. So, at this point, GamaSlicer contributes to proof such concepts.

The Java class in Listing 1 contains a method to implement the above stated MinMax problem, complemented with JML annotations to described the specific user needs.

Listing 1. Annotated Method — MinMax

```
public class MinMax {
   private int absmax, absmin;

   public MinMax() {}
    /*@
      @ requires a > 0 && b > 0;
      @ ensures ((absmax == a) && (absmax >= b)) ||
      @         ((absmax == b) && (absmax >= a));
      @*/
    public void MinMaxFunction(int a, int b) {
        if ((a > 0 && b > 0) || (a < 0 && b < 0)) {
            if (a>0) {
                if (a > b) {
                    absmax = a; absmin = b;
                }
                else {
                    absmax = b; absmin = a;
                }
            }
            else {
                if (a > b) {
                    absmax = -b; absmin = -a;
                }
                else {
                    absmax = -a; absmin = -b;
                }
            }
        }
        else {
            absmax = 0; absmin = 0;
        }
    }
}
```

Observing carefully the precondition and postcondition listed above, it clearly specifies that the programmer is only concerned with the computation of the *maximum* of two *positive* numbers.

As a file with that class in Listing 1 was previously included in our repository, to start playing with GamaSlicer, just pick MinMax from the list and upload it. The Java+JML FE is then automatically invoked to parse the source code and build the AST and the Identifiers Table. After a successful recognition, the web interface Tabs display the following information:

- The source program is displayed in Tab 1 – in a pretty print manner, with highlight syntax and line numbers (Figure 1); also the buttons to activate the SMT code-generator or the assertion-based slicing algorithms are included in the top line of Tab 1 (notice that in Figure 1).
- The AST in Tab 2;
- The Identifiers Table in Tab 3;
- The LCFG for the source program in Tab 6 (Figure 6).

Tab 3 also displays for each identifier additional information like its category (variable, class/method name, etc.), its type, the owner, and its status (public, static, final). In case of constants (identifiers that are both static and final), their value is also shown.

Now we are able to execute step-by-step the VCGen algorithm. Each time the algorithm is executed, the rule applied appears on the right of the source code (Figure 2). As referred previously, the rules are displayed in a tree-form, in order to make easier the process comprehension. Using this representation, the recursion of the algorithm becomes evident. Besides that, it is possible to inspect with which arguments the rule was applied for (Figure 3). This feature is useful when we do not understand the results obtained in some generation step and we want to check them. For instance, in Figure 3, we are trying to figure the arguments of the rule

$$wp(C1; C2, Q)$$

was invoked. This allows us to comprehend the order of the arguments when dealing with a sequence of instructions (that corresponds to a recursive call). In this first occurrence the result is:

```
C1: absmax = a
C2: absmin = b
Q: absmax == a && absmax >= b ||
   absmax == b && absmax >= a
```

Invoking the same rule, but some steps after, we are able to understand that the arguments are different:

```
C1: absmax = b
C2: absmin = a
Q: absmax == a && absmax >= b ||
   absmax == b && absmax >= a
```

Once the rules were all applied, the button in Tab 1 that allows the SMT code generation is enabled. After requesting SMT code generation, Tab 5 displays the generated code and we can now verify it.

After code verification, a table with the prover results is displayed (Figure 4). Lines in the table show which prover was used and columns which formula was evaluated. After the prover terminates to proof the formulae, the time spent on this task is also shown. This is particularly useful if we aim at understanding which prover is more efficient and in which circumstances.

We are willing to combine different provers that accepts SMT as input (e.g. Z3, Yices, Alt-Ergo, CVC3), but unfortunately some syntax issues still need to be solved (only Z3 accepts all inputs).

If alternatively we chose the slicing button in Tab 1 instead of the SMT Generation button, the result of applying the assertion-based slicing algorithm will be displayed in Tab 5. Lines that can be removed from the original program appear strike-out and in red color. Besides that, at top of the program, a list of such lines is displayed (Figure 5).

After explaining the idea behind these verification and slicing steps and their visualization, the demonstration will illustrate the animation capabilities of GamaSlicer. For that

purpose, let us focus again on Figure 6. To animate we will highlight the branches (edges or paths), one by one, in order to mark the assertion-pair that is being tested; we use a color code that depends on the logical values returned by the theorem-prover. Each time the implication labeling a branch evaluates to true, the statements between the enter-exit nodes of that branch can be cut off. As a new picture is generated (over the same graph layout, fixed due to a special algorithm we use) after each proof, this creates the desired verification animation. At the end a shortest-path algorithm is used to determine the minimum number of statements that will remain (all the other are sliced out).

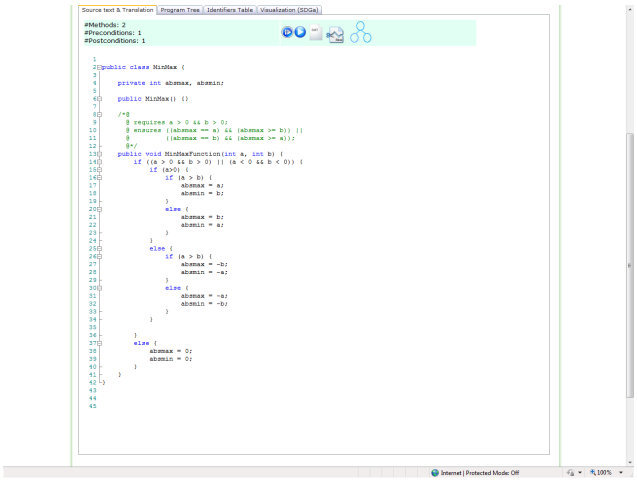


Figure 1. [Tab 1] Source code

REFERENCES

[1] D. da Cruz, P. R. Henriques, and J. S. Pinto, "Contract-based slicing," http://alfa.di.uminho.pt/~danieladacruz/FLACOS2010contractSlicing.pdf, April 2010.

[2] J. B. Barros, D. da Cruz, P. R. Henriques, and J. S. Pinto, "Assertion-based slicing and slice graphs," http://alfa.di.uminho.pt/~danieladacruz/SEFM2010specSlicing.pdf, March 2010.

[3] D. da Cruz, P. R. Henriques, and J. S. Pinto, "Gamaslicer: an online laboratory for program verification and analysis," in *Proceedings of the 10th Workshop on Language Descriptions Tools and Applications (LDTA'10)*, 2010.

[4] M. J. Frade and J. S. Pinto, "Verification conditions for source-level imperative programs," Universidade do Minho, Tech. Rep. DI-CCTC-08-01, 2008.

[5] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon, "Program slicing based on specification," in *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2001, pp. 605–609.

[6] J. J. Comuzzi and J. M. Hart, "Program slicing using weakest preconditions," in *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*. London, UK: Springer-Verlag, 1996, pp. 557–575.

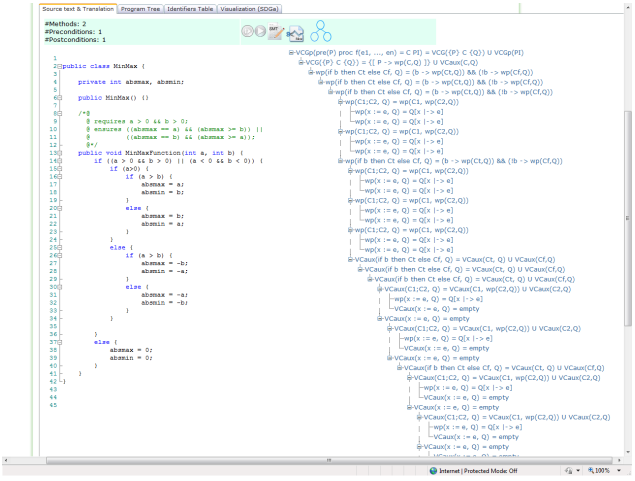


Figure 2. [Tab 1] Source Program and VCGen Rules applied

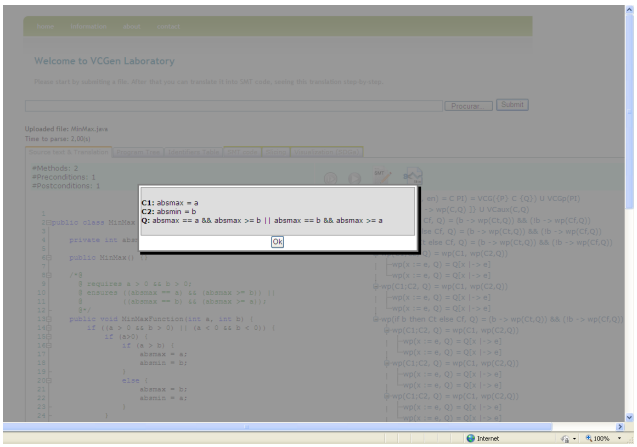


Figure 3. Inspecting the arguments which the VCGen algorithm was applied.

Figure 4. [Tab 4] SMT code verification

Figure 5. [Tab 5] Sliced Program

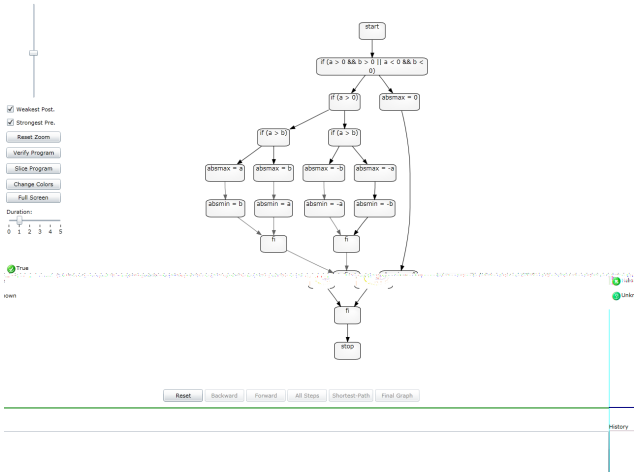
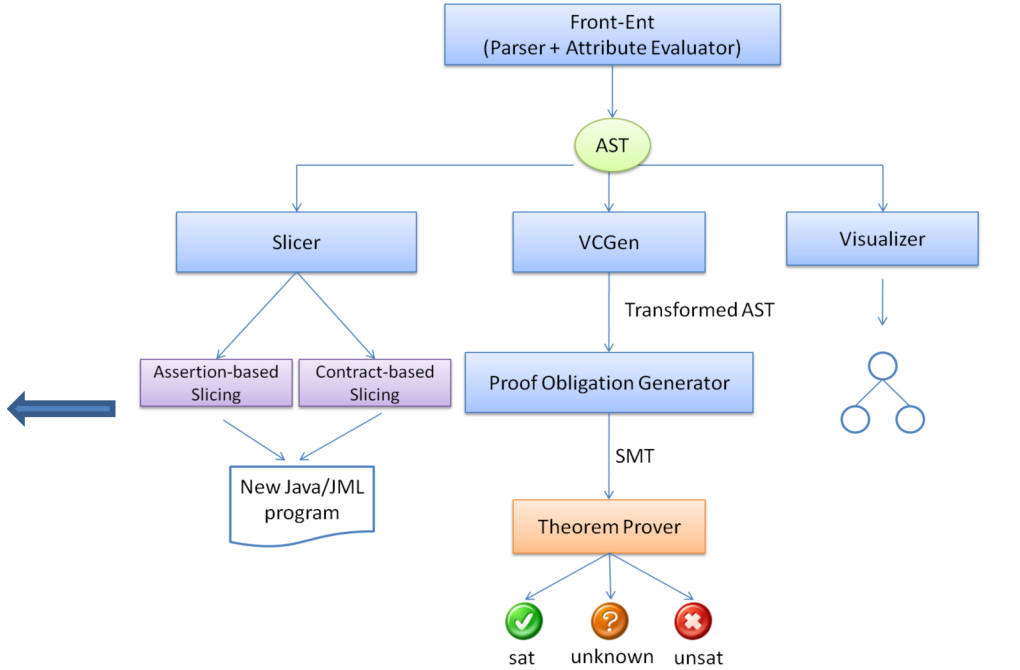
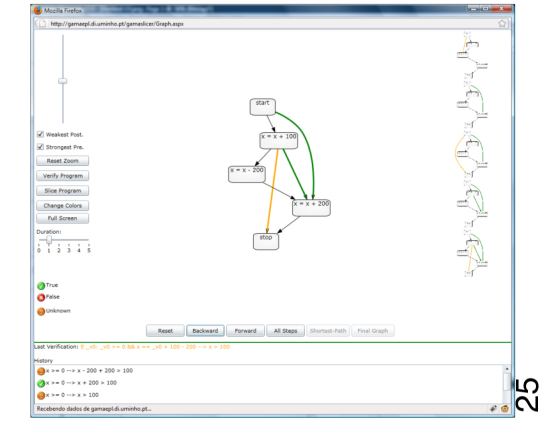
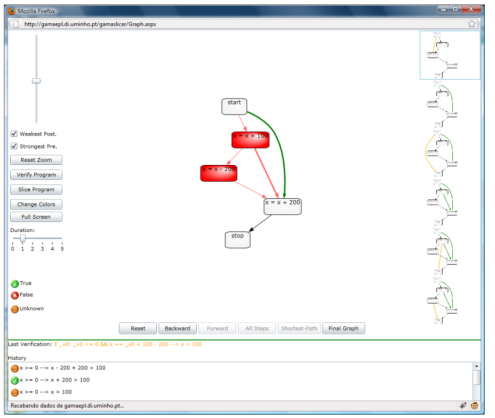


Figure 6. [Tab 6] LCFG Visualizer

A tool to verify, slice and animate annotated components: GamaSlicer







Daniela da Cruz, Pedro Rangel Henriques, Jorge Sousa Pinto
Departamento de Informática / CCTC, Universidade do Minho, Portugal

A Preliminary Exercise of a Database Application Verification using *LinguSQL*

Ade Azurat, Rikky Wenang Purbojati, Api Perdana, Heru Suhartanto
Faculty of Computer Science
University of Indonesia
Depok, Indonesia
{ade,heru}@cs.ui.ac.id

Abstract— LinguSQL is an experimental development tool that integrates both blackbox testing and whitebox code verification during the code-writing activity. This research explores the application of LinguSQL in the development of a stock transaction application tool. This preliminary exercise is expected to bring forward an advantage on the overall testing and verification phase, and in the context of tool implementation, expose the parts that need further improvement for real world application.

Keywords; Formal Method; case study; database application verification; tool support.

I. INTRODUCTION

The applications of database-enabled information systems have been affecting our everyday business. These systems have to be designed in accordance to the expected functionality as well as using the correct algorithms. Failure to do so can result in financial as well as non-financial losses.

LinguSQL [1,2] is designed to verify and validate an algorithm in a *Lingu* script using LinguHOL library[3]. The Lingu Language and its LinguHOL library were develop by Prasetya[4]. *Lingu* is a light and abstract programming language that emphasizes on the operation of data transformations in database. *Lingu* does not possess all the functionality of a common programming language such as Java, but has sufficient syntax and rules that enables us to design an algorithm to manipulate data in a database. The *Lingu* language and logic has been embedded in the *Higher-Order Logic (HOL*) theorem prover. This embedding is called *LinguHOL* library.

We explores the use of *LinguSQL* for a case study of an application that handles stock transactions in Indonesia[2]. The algorithm of stock transactions are implemented as *Lingu* scripts. It will then be verified, validated and transformed to a more popular programming language using *LinguSQL*.

II. LINGUSQL

LinguSQL is a tool to run verification and validation of a predefined algorithm and specification which then can be transformed into an executable code. One has to prepare a defined algorithm of a problem and its test specification prior to using this tool. The defined algorithm in Lingu can utilize a set of native database transformation operator inside its method or function. Moreover it also provide a set of logical syntax to provide precondition and postcondition specification for those methods or functions. Consequently, verification and validation process will be run against these predefined specification.

III. FEATURES

LinguSQL is still in prototyping phase. The expected complete features will be the following:

Integrated Testing Environment. One of the main feature of LinguSQL is its integrated testing environment for the development process using Lingu abstract language. LinguSQL is intended to produce a safe and more reliable code inside a software package by running it against both whitebox and blackbox testing. Using this feature, defects and errors are expected to be lower compared to those development process using only whitebox or blackbox testing.

The tool provide an integrated whitebox testing environment to developer using Higher-Order Logic (HOL) theorem prover. HOL theorem prover is needed since most Lingu's syntax is in the form of higher-order logic where its operator have some kind of quantification informations on them. Using the theorem prover, LinguSQL first break the algorithms and specifications into a set of mathematical prepositions called verification conditions. Afterwards, these verification conditions are sent to a Higher-Order Logic (HOL) theorem prover to assess its correctness property.

In addition, LinguSQL provides a validation means for developer to test their algorithm blackbox-wise. Since validation is a process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model [5], LinguSQL is able to takes a predefined scenario and executes it with a set of data which resembles real world data. This data is made available by LinguSQL by generating it using internal random data generator. Afterwards the output of the test can be compared with the predefined postcondition specification to assess if it is the expected outcome of the function execution.

Code Generation. Another main feature of LinguSQL is the ability to transform given Lingu script into another programming code. Current version of LinguSQL is able to transform Lingu script into a compilable and executable Java

code. Transformation process takes place when Lingu script has been verified and validated. This guarantee that the transformation result will inherit the same nature properties as the abstract language it is derived from.

Modular Transformation System. Each language has its own characteristics and uniqueness. System design that tries to handle all transformation of the language in one big application is inefficient. This leads to the design that support plug-in style module for each different language. The desired effect of the design is that this tool has an improved scalability against a whole array of programming language. One can write a transformation module for C# code independently or even write a transformation module for newer language such as Ruby.

Internal Data Generator. The execution of scenario testing demands an environment that resembles real world. This includes the data that are being used. To accommodate the needs, LinguSQL provides an automatic test generation. The automatic test generation is able to generate a collection of data that resembles the actual data. This is done by using a constraint for the data that are being generated. For instance, if we generate a salary data, then some constraints are needed to make the salary data looks valid. Such as salary data cannot be negative or zero, or that the sum of all salary cannot exceed the corporation salary budget. Ignorance of this issue will degrade the credibility of the verification and potentially will create problem when it is implemented in actual use.

Interactive Proving. LinguSQL provides an interactive means to aid HOL theorem prover to prove the verification conditions. By default HOL theorem prover tries to verify these conditions automatically. However because of the nature of HOL is undecidable, sometimes HOL theorem prover cannot solve the problem on its own. It needs an expert assistance to guide and tell HOL backend what tactics or lemma to be used. This feature ensures that a human intervention can be done anytime HOL theorem prover meets some unexpected obstacle in proving the verification conditions.

IV. Workflow

In a broad outline, the process of developing and testing of an application using *LinguSQL* is comprised of 4 stages:

1. Defining data and function. The data, functions and methods that will be involved in the application's problem domain are identified and defined as tables in the database.
2. *LinguHOL* verification. This stage ensures the flow correctness of an algorithm of a procedure contained in the *Lingu* script with the help of HOL theorem prover.
3. *Lingu* script validation. This stage tests whether the execution of procedures that have been defined in the first stage will produce the expected output.
4. Transformation to another language. The last stage of development is the transformation from *Lingu* script to another programming language, such as *Java*.

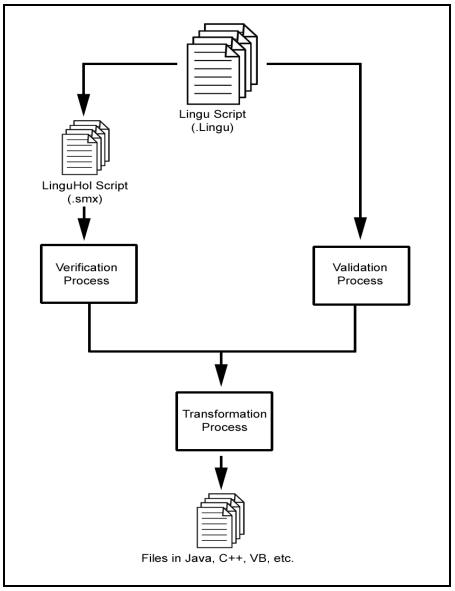


Figure 1. Development Flow of *LinguSQL*

The expected result of the development using *LinguSQL* is some components of code that has been thoroughly verified.

V. Stock Exchange Case Study

Aside from the generally accepted procedures in the stock market, a stock exchange information system has to consider local capital market regulations, transaction periods, or limiting the number of transaction per second.

As a case study, the above rules can be implemented in *LinguSQL* to ensure that algorithm in the stock exchange information system will not violate those rules.

The stock exchange system will be greatly simplified in this preliminary research. The system will only include recording of requests to buy, recording of offers to sell, and a mapping process to match offers with requests.

VI. Lingu Script Implementation

The first step is to define the data that will be used in stock transactions. The next step is to form the transaction mapping rules and other supporting rules as algoritms in *Lingu*. One of the rules is: "*The transaction of invalid stock checking algorithm does deletion of offers and requests of invalid stocks, such as if the stock is not registered in the system or stocks that are suspended.*". Figure 3 shows the *Lingu* script of the rule.

```
method filterInvalidStockTransaction () :: ()

ok_ids,ids :: Table {| Id :: Integer; |};

do {
ids := findAll d<-database.BuyTransactionTab
       where T found d.Id;

ok_ids:= findAll d<-database.BuyTransactionTab,
         b<-database.StockTab
         where d.Stock_Id == b.Id found d.Id;

delete ids where ids.Id in ok_ids.Id;

insertAll d<-database.BuyTransactionTab,i<-ids
          where d.Id == ids.Id
          to database.InvalidTransactionTab;
}
```

Figure 2. A Lingu Script example of the case study

The *LinguSQL* system will generate a verification condition based on the given *Lingu* script and pre-post specifications given by the programmer. Figure 3 shows the generated verification conditions and some instructions to call the implemented decision procedures in *LinguHOL* to automatically prove it in *HOL* theorem prover environment.

```
val filterInvalidStockTransaction_def =

Define `filterInvalidStockTransaction(

REF(StockTab: StockListTable set),
REF(BuyTransactionTab: TransactionTable set),
REF(InvalidTransactionTab: TransactionTable set)
)
=

pre  (empty InvalidTransactionTab
      /\ ~(empty StockTab)
      /\ ~(empty BuyTransactionTab) )

post ( ALLof InvalidTransactionTab
       (satisfy r. ALLof StockTab
         (satisfy q. ~(r.Stock_Id = q.Id))))

do /{
     let
        badids = select BuyTransactionTab I
        (only r. ALLof StockTab(satisfy t.
        ~(t.Id = r.Stock_Id)))
     in
         insert badids InvalidTransactionTab I
ALL
/}
return void`;

(*----- verification -----*)
reduce defs filterInvalidStockTransaction_def ;
L0min_vcg.autoverify MY_TAC ;
L0min_vcg.VCs;
L0min_vcg.conclude();
```

Figure 3. A generated verification condition.

VII. Concluding Remarks

This case study was used because of its dependence on database operations and its mission-critical properties, resulting in the need of a thorough testing or verification in order to prevent financial losses. The case study was implemented in accordance to a simplified version of Indonesian Stock Exchange Regulation. The writing of the *Lingu* script in this case study has produced concrete components in Java which can be integrated to the bigger system of the recording of stock exchange transactions.

A few shortcomings that arose from the implementation of this case study is the limited *Lingu* conditional expression in the *update* and *select* operation, and the rudimentary application used to transform *Lingu* to *LinguHol*. The return value of a procedure call is yet to be implemented. These mentioned weaknesses have to be dealt with before *LinguSQL* can be used to develop real world applications.

More detail information of *LinguSQL* is available in:

http://fmse.cs.ui.ac.id/?open=lingusql/index

References

[1] R. Wenang, I.S.W.B. Prasetya, S. Maizir, B. Wibowo, & A. Azurat, "LinguSQL: A Verification and Transformation Tool for Database Application". In Proceedings of 6th National Seminar of Computer Science and Information Technology (SNIKTI) Indonesia 2005.

[2] Rikky Wenang Purbojati, Ade Azurat, Api Perdana, Heru Suhartanto, "*Studi Kasus LinguSQL: Aplikasi Transaksi Perdagangan Saham*", Journal of Information Systems, Vol. 5 – No. 1, April 2009, Faculty of Computer Science University of Indonesia. ISBN 1412-8896.

[3] I.S.W.B Prasetya, A. Azurat, T. Vos, and A.v Leeuwen. "Incremental implementation of syntax driven logics". Journal of Software, 1(3):1–13, September 2006.

[4] A. Azurat, I.S.W.B. Prasetya, T.E.J. Vos,H. Suhartanto, B. Widjaja, L.Y. Stefanus,R. Wenang, S. Aminah, J. Bong. "Towards Automated Verification of Database Scripts". In Proceedings of 18th International Conference on Theorem Proving in Higher Order Logics 2005.

Appendix: LinguSQL Workflow and Screenshot

LinguSQL basically requires 4 main steps in its workflow to produce a compilable and executable code.

The first step of the workflow is the loading of Lingu script. This step will load the defined algorithm and specification into LinguSQL memory space to be used on the verification, validation, and transformation process. Figure A.1 shows LinguSQL that has been loaded with a Lingu script.

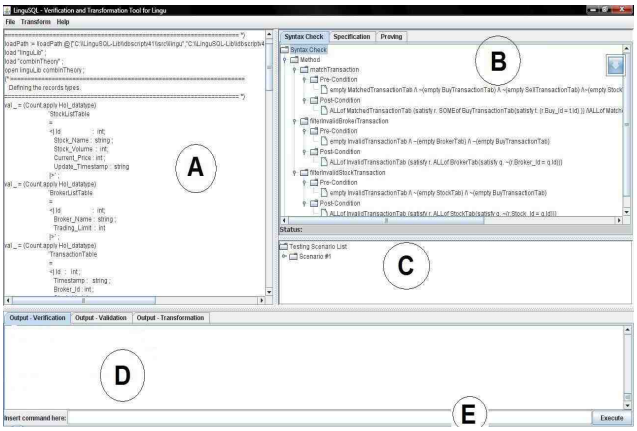


Figure A.1. LinguSQL loaded with Lingu script.

Box A on Figure A.1 contains the whole Lingu script for the purpose of reviewing and editing. LinguSQL parses this script and put all of the precondition and postcondition specifications onto box B. Meanwhile scenario for validation process is put on box C. Results of every workflow steps will be shown on the box D. Specific for the verification process, box E will be used to provide interactive proving assistance to the HOL theorem prover.

The next step is verification process which utilize HOL theorem prover to assess its specification correctness. Figure A.2 shows the result interface after the verification process is done.

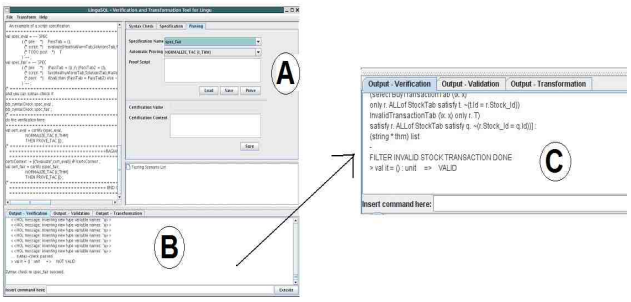


Figure A.2. Verification Result of LinguSQL.

One can proves the script by selecting the specification listed in Figure A.1 box B. After he selects the desired specification, he can utilizes a set of tactics and proving techniques as seen on Figure A.2 box A. The result of verification process for certain specification is shown in the box C on Figure A.2. Successful verification enables developer to continue to the next step.

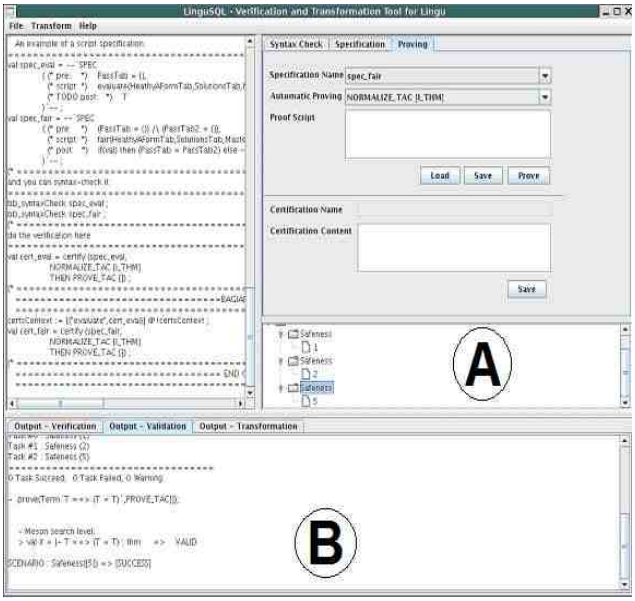


Figure A.3 Validation Result of LinguSQL.

The validation steps uses scenario that are listed on Figure A.3 box A. The validation box contains the whole scenario specified inside the Lingu script, which also shows their parameters. The scenario "safeness" with parameter of 5 will execute using the internal data generator to check if the scenario meets the specification defined previously. The result of the selected scenario test can be seen on Figure A.3 box B. If the whole validation tests are successfully run, one can process in generating the desired code to be compiled or executed.

Finally the last step of this development process is code generation or transformation. Current LinguSQL version is able to generate a Java code based on the given Lingu script.The transformation or code generation process result can be seen on the output box A on Figure A.4.

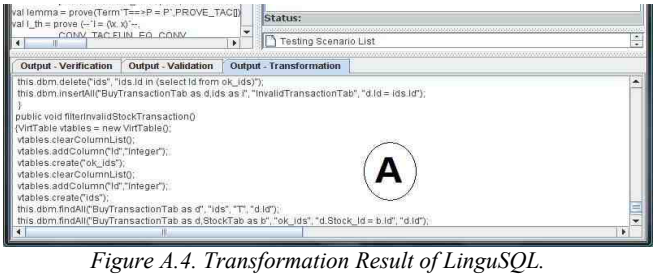


Figure A.4. Transformation Result of LinguSQL.

Latest information of *LinguSQL* is available in:

http://fmse.cs.ui.ac.id/?open=lingusql/index

LinguSQL:
An IDE for Reliable Database Application

A. Azurat, I.S.W.B. Prasetya, T.E.J. Vos, H. Suhartanto, B. Widjaja, L.Y. Stefanus, R. Wenang, S. Aminah, J. Bong.

http://fmse.cs.ui.ac.id/?open=lingusql/index

What is Lingu?

A light weight language to program data transformation on database (in SQL) integrated with verification and formal testing .

➢ Domain Specific Language.
➢ Abstract Language.
➢ No optimization features (e.g. No Arrays nor pointer).
➢ Small. Limited constructs (e.g. no String[n] nor sort by modifier).

What is LinguHOL?

Lingu's Programming Logic embedding in HOL theorem prover.
➢ Generates proof obligations.
➢ Certification for procedure call.
➢ (Semi) Automatic reduction (NORMALIZE_TAC) for lingu's SQL statements.

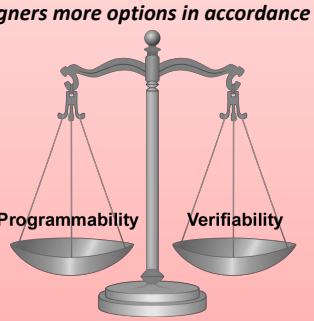
Soundness and Decidebility
✔ Lingu logic is **sound.**
✔ Lingu specifications are **decidable.**

Example (*ScriptShot)*

```
> val safe_def =
   - !HealthyAFormTab SolutionsTab MasterTab PassTab val dummy.
       safe (HealthyAFormTab,SolutionsTab,MasterTab,PassTab,val) dummy =
       call
         (evaluate (HealthyAFormTab,SolutionsTab,MasterTab,PassTab)
            dummy) >>
       (let
          val1 =
          forall PassTab
            (\r.
              exists MasterTab (\t. r.ID = t.ID) /\
              exists HealthyAFormTab (\y. r.ID = y.ID)) in val ::= val1) :
thm

> val spec_safe =
   `SPEC
     ((PassTab = {}) /\ (val = F),
      safe (HealthyAFormTab,SolutionsTab,MasterTab,PassTab,val) dummy,
      forall PassTab
        (\r.
          exists MasterTab (\t. r.ID = t.ID) /\
          exists HealthyAFormTab (\y. r.ID = y.ID)) =
       val)` : term
-
> val cert_eval =
   L0logic.CERT(`SPEC
                  (PassTab = {},
                    evaluate (HeathyAFormTab,SolutionsTab,MasterTab,PassTab)
                    dummy,T)`,
                |- (PassTab = {}) ==> T) : CERT

- val cert_safe = certify (spec_safe,
                   NORMALIZE_TAC [I_THM]
                   THEN PROVE_TAC []);

.... syntax-check passed.

> val cert_safe =
  L0logic.CERT(`SPEC
               ((PassTab = {}) /\ (val = F),
               safe (HealthyAFormTab,SolutionsTab,MasterTab,PassTab,val) dummy,
               forall PassTab
                 (\r.
                   exists MasterTab (\t. r.ID = t.ID) /\
                   exists HealthyAFormTab (\y. r.ID = y.ID)) =
                 val)`,
             |- (PassTab = {}) /\ (val = F) ==>
             (PassTab = {}) /\
             (T ==>
              (forall %%genvar%%9469
                (\r.
                   exists %%genvar%%9468 (\t. r.ID = t.ID) /\
                   exists %%genvar%%9466 (\y. r.ID = y.ID)) =
                   forall %%genvar%%9469
                 (\r.
                   exists %%genvar%%9468 (\t. r.ID = t.ID) /\
                   exists %%genvar%%9466 (\y. r.ID = y.ID))))) : CERT
```

Vision

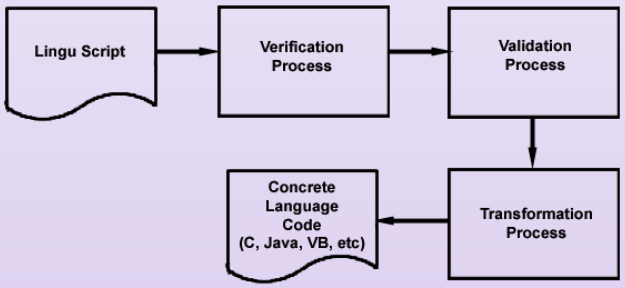
Our view towards program correctness is a pragmatic one. We define the goal of programming as to make a product where the balance between reliability, risk, and cost is decided consciously. We aim to develop a tool that can support verification in a stratified way, ranging from low-cost light verification to extensive verification, so as to give designers more options in accordance to available budget.

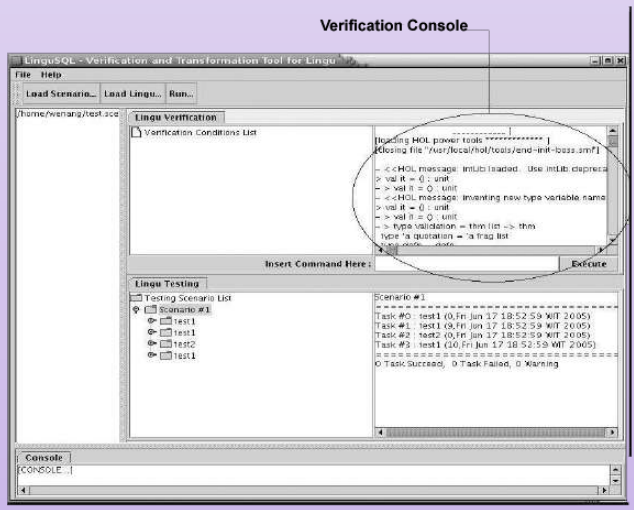


Our design philosophy is not to support an overly rich target programming language, as such requires a complicated logic which is difficult to maintain. Since verification is often much more expensive than the development of a program itself, we prefer a target language with a more balanced focus between programmability and verifiability.

What is LinguSQL?

Lingu's IDE (Integrated Development Environment) for reliable database application.
➢ Code Transformation from Lingu to Java.
➢ Formal program verification with theorem prover (HOL) support for database application.
➢ Integrated testing with sample data generator. Tester should only list the scenario to be tested.





A Petri Net-based Editor to Develop Active Rules

Lorena Chavarría-Báez
Departamento de Posgrado
Escuela Superior de Cómputo - IPN
México, D.F.
lchavarria@computacion.cs.cinvestav.mx

Xiaoou Li
Departamento de Computación
Centro de Investigación y de Estudios Avanzados - IPN
México, D.F.
lixo@cs.cinvestav.mx

***Abstract*—Active rules are a powerful mechanism to represent reactive behavior. In this paper we describe a software tool that creates active rules from a Petri net structure and automatically verifies the rule base, i.e., it finds structural errors such as *redundancy*, *inconsistency*, *incompleteness* and *circularity*.**

***Index Terms*—Active rule, verification, CCPN**

I. Introduction

Active rules respond to relevant events for an application by executing certain procedures related either to the system or to the environment. Smart homes, and active databases [4] introduce them into the system to manage some of their important activities. An active rule consists of an *event*, that occurs inside or outside the system and *triggers* the rule, a *condition*, that checks the triggering context and *fires* the rule if it was satisfied, and an *action*, which specifies the task to be executed once the event happened and the condition was evaluated to true.

Developing an active rule base (ARB) is not an easy task. It involves acquiring knowledge and incrementally developing the rule base until its completion. Additionally, the rule designer has to use existing available tools to solve the problem. Nevertheless, tools associated with an active system may be minimal, with little support for browsing, monitoring, or debugging of active rules. References [3], [2] present editors able to create rules dynamically and to support complex event processing applications, respectively. However, they don't perform complete tasks of error detection.

We introduce a Petri net-based editor of active rules, able to create active rules through the construction of a Petri net structure. Our editor allows non-expert users specify rules using simple graphical elements and then formalism of Petri nets is used to perform a formal evaluation of the rule base. Finally, users can obtain an error-free rule base with no need to deeply know formal methods.

II. Active Rules

The general form of an active rule is:
ON *event*
IF *condition*
THEN *action.*

An *event* is something that occurs at a point in time. It can be of two types: *primitive* or *composite*. An event is primitive when it cannot be decomposed in smaller events, for example, a transaction in database operation. An event is composite when it is defined by some combination of primitive or composite events using a range of operators that constitute the event algebra [4]. The *condition* examines the context in which the event has taken place. The *action* describes the task to be carried out by the rule if the condition is fulfilled once an event has taken place.

When an active rule base is developed some errors may be unnoticed introduced, for example, redundancy, inconsistency, incompleteness and circularity. Those anomalies must be detected before the system starts working to prevent abnormal behavior.

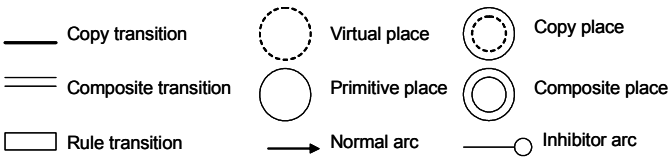


Fig. 1. List of CCPN elements

III. Design and Implementation of the Editor

Our editor allows user to create an active rule base from the graph of a Petri net extension called Conditional Colored Petri Net (CCPN). Then formalism of CCPN is used to detect errors.

CCPN was especially developed to represent each element of active rules, their execution and interaction [1].

A. Conditional Colored Petri Net

Figure 1 shows the basic graphical elements of CCPN.

In CCPN, an active rule is mapped to a transition where its condition is attached, event and action parts are mapped to input and output places of the transition, respectively (Figure 2(a)). Matching between events and input places has the following characteristics (see Figure 1):

1. *Primitive* places, represent primitive events;
2. *Composite* places, represent composite events;
3. *Copy* places, are used when one event triggers two or more rules. A copy place takes the same information as its original one;

Rules and transitions are related in the following form (see Figure 1):

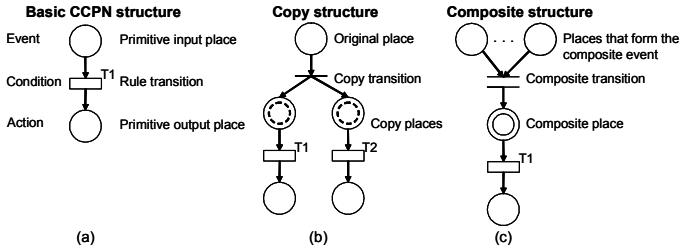


Fig. 2. Basic CCPN structures of an active rule

1. *Rule* transitions, represent rules;
2. *Composite* transitions, represent composite event generation;
3. *Copy* transitions, duplicate one event for each triggered rule.

Whenever an event triggers two or more rules it has to be duplicated by means the copy structure depicted in Figure 2(b). Composite transition's input places represent all the events needed to form a composite event while its output place correspond to the whole composite event (Figure 2(c)). The CCPN model of a set of active rules is formed by connecting those places that represent both the action of one rule and the event of another rule. Since a rule typed transition corresponds to a rule in the rule base, we use transition or rule indistinctly.

CCPN is easily manipulated using its mathematical representation given by incidence matrix, $A_{m\times n}$. Incidence matrix shows the flow relations between places and transitions. Rows of the matrix represent transitions and columns represent places. The value of an element $A_{i,j}$ represents the weight of the arc that connects the transition i with the place j. If this value is zero, there is not connection between the transition i and the place j. If it is negative, place j is an input place for the transition i. Finally, if it is positive, place j is an output place for the transition i. We denote as $A^{+}_{i,j}$ the positive entry located in $i-th$ row and $j-th$ column, and as $A^{-}_{k,l}$ the negative entry located in $k-th$ row and $l-th$ column.

B. Architecture

Our editor was developed in Java and under *ECAPNSim* architecture [1] (see Figure 3). Normally, the module *Rule editor* allow user to write a set of active rules with the general syntax. Then, module *ECA - CCPN converter* takes this file and automatically generates its corresponding CCPN model as well as its incidence matrix. Finally, the module *CCPN editor/visualizer* shows the CCPN generated. Some environment tools are provided to perform a formal evaluation of the rules, for example, the module *Rule verification* uses the incidence matrix to produce a list of errors that the active rule base contains, also it generates a list of possible corrections.

In our approach we proceed conversely, i.e., we let the user draws places, transitions and arcs to connect them and

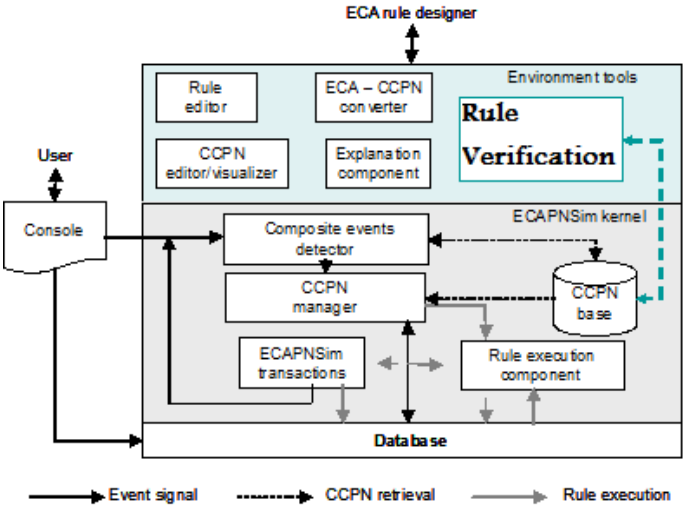


Fig. 3. ECAPNSim's architecture

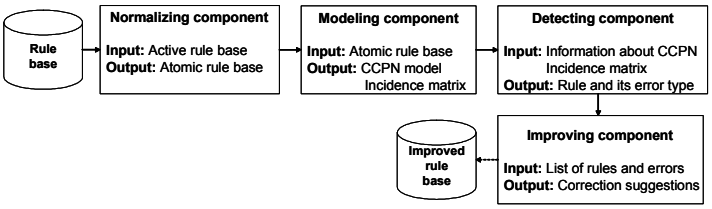


Fig. 4. Main components of *ECAPNVer*

to form a CCPN structure. During the construction of the CCPN, we use some of the tools that we have developed to check if the structure of the CCPN is right and to identify some errors, for example, if a cycle is introduced, a new window is opened to alert the user. At the end of the process, we ask the user for specific information which will be used to create the final rule base. When the semantics of the rules has been captured, the module *Rule verification* works to find errors and alert the user. If there is no error in the rules, a text file describing them is created.

Verification of the active rule base consists of four major phases which are illustrated in Figure 4.

1) Normalizing.
   The Normalizing component takes as input an active rule base, then it transforms each rule of the rule base into an atomic rule. The output of the Normalizing component is a rule base containing atomic active rules. An atomic rule is that whose event and condition are a conjunction of one or more primitive events and conditional clauses, and its action is only one instruction. This process is done to simplify later analysis.
2) Modeling
   The Modeling part has two main algorithms. The first one takes the atomic active rule base produced by the Normalizing component and generates the

corresponding CCPN model. During this phase, relevant information can be obtained from the CCPN. The second algorithm takes the CCPN graphical model and obtains its mathematical model given by the connection matrix.

3) Detecting.
This is the most important part of the *Rule Verification* module. It is composed of two parts: *Detection* and *Interaction Analysis*. *Detection* uses the incidence matrix of the CCPN as well as information generated during CCPN construction, for example, type of transitions and places, to perform error detection. *Detection* produces a list of possible anomalous rules. *Interaction Analysis* takes the list produced by the previous component and analyzes the way in which they interact, i.e., it checks activation, deactivation, commutativity, and so on. In this step, some of the previous rules detected as possible anomalous can be ruled out. The output of this element is a list of those rules which have redundancy, incompleteness, inconsistency and circularity problems.
4) Improving.
The task of the Improving component is to take the list of anomalous rules identified by the Detecting element and to formulate a set of improvement suggestions taking into account the type of problem. The improvement suggestions are displayed in order to active rule designer can (if it is necessary) sketch the active rule base again.

C. CCPN Editor

The main screen of the editor is showed in Figure 5. The editor has basic elements: place (circle), transition (rectangle), arc and token (filled circle). This last one is used to perform simulation and explanation of rule behavior. It also has a Property window to capture information of rule elements.

In order to create a rule, user has to drag the elements into the panel and if they are wrong connected a message is sent to he/she. If the user introduces a structural error, for example, a loop, the system also is able to detect it.

IV. Example

We will use our editor to create a small set of active rules for a database application. We draw the primitive places, rule transitions and corresponding arcs of the CCPN showed in Figure 5. The copy structure was added by the system automatically to perform rule simulation. Then, the system asks us, using the Property window, for information about places and transitions.

Figure 6 shows the final rule base.

V. Conclusion

Developing active systems is promising; however, challenging since tools associated with them may be minimal.

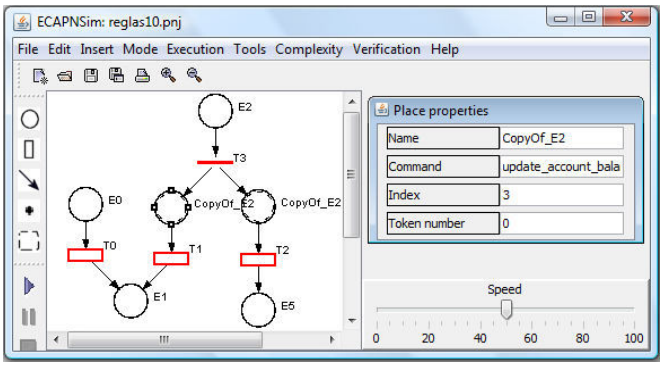


Fig. 5. CCPN editor. T3 is automatically generated

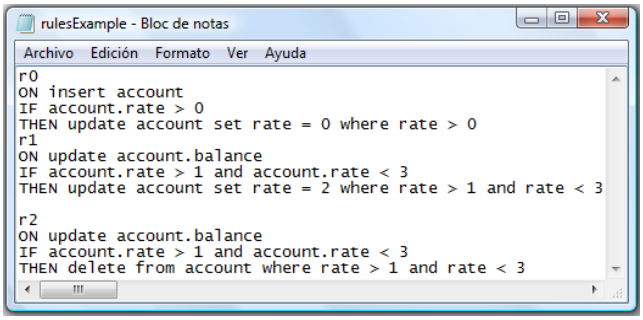


Fig. 6. Rules of the CCPN in Figure 5

In this paper we describe a tool able to create a rule base from a CCPN structure. We argue that our tool can be an aid to bring formal methods to non-experts in an easy way and it can improve the quality of active systems since it also performs formal evaluation on the rule base.

References

[1] X. Li, J. Medina-Marín, and Chapa S., "Applying Petri Nets on Active Database Systems", *IEEE Trans. on System, Man, and Cybernetics, Part C: Applications and Reviews,* Vol. 37, No. 4, pp. 482 - 493, 2007.

[2] A.M. Ericsson, and M. Berndtsson, REX, the Rule and Event eXplorer, in *Proc. of the 2007 inaugural international conference on Distributed event-based systems*, Toronto, Ontario, Canada, vol. 233, pp. 71 - 74, 2007.

[3] S. Chakravarty, and S. Varkala, Dynamic Programming Environment for Active Rules, in Proc. of the 7th Intl. Baltic Conference on Databases and Information Systems, pp. 3 - 16, 2006.

[4] N. Paton, O. Díaz, Active Database Systems, *ACM Computing Surveys*, Vol. 31, No. 1, pp. 62-103, 1999.

[5] S. Comani, L. Tanca. Termination and Confluence by Rule Prioritization, *IEEE Trans. on Knowl. and Data Eng.*, Vol. 15, No. 2, pp. 257-270, 2003.

[6] E.Baralis, J. Widom, An Algebraic Approach to Static Analysis of Active Database Rules, *ACM Trans. on Database Systems*, Vol. 25 , Issue 3, pp. 269 - 332, 2000.

[7] J. C. Augusto, and C. Nugent, A New Architecture for Smart Homes Based on ADB and Temporal Reasoning, in *Toward a Human Friendly Assistive Environment (Proc. of 2nd Intl. Conf. on Smart homes and health Telematic, ICOST2004), Assistive Technology Research Series*, Vol. 14, pp. 106-113, IOS Press, Singapore, September 15-17, 2004.

A Petri Net - based Editor to Develop Active Rules

Lorena Chavarría-Báez
Xiaoou Li

I. Introduction

Active rules respond to relevant events for an application by executing certain procedures related either to the system or to the environment.

Examples:

* Smart homes
* Active databases

Problem: tools associated with an active system may be minimal, with little support for browsing, monitoring, or debugging of active rules.
References [2], [3] present editors able to create rules dynamically and to support complex event processing applications, respectively. However, any of them performs tasks of error detection.

II. Active Rules

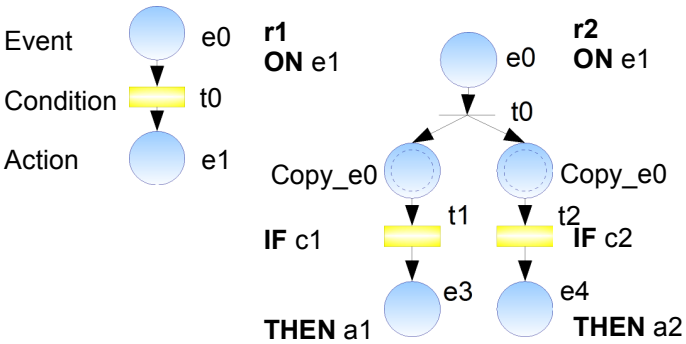
Syntax:

ON *event*	occurs inside or outside the system and *triggers* the rule.
IF *condition*	checks the triggering context and *fires* the rule if it was satisfied.
THEN *action*	specifies the *task* to be executed once the event happened and the condition was evaluated to true.

III. Design and Implementation of the Editor

Based on the Conditional Colored Petri Net (CCPN) [1].

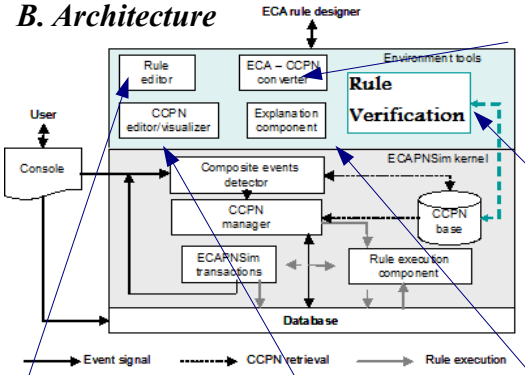
A. Conditional Colored Petri Net



a. Basic CCPN structure b. Copy structure

Figure 1. Basic CCPN structures of an active rule

B. Architecture



generates a CCPN model and its incidence matrix

produces an error list of a rule base, it also generates a list of possible corrections.

allow user to write a set of active rules/generates a rule base from a CCPN

shows the CCPN/ draws CCPN elements

performs rule behavior simulation.

Figure 2. ECAPNSim Architecture

C. CCPN Editor

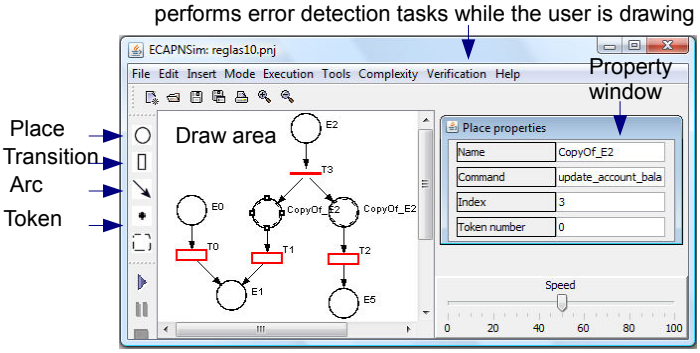


Figure 3. CCPN editor. T3 is generated automatically

IV. Example

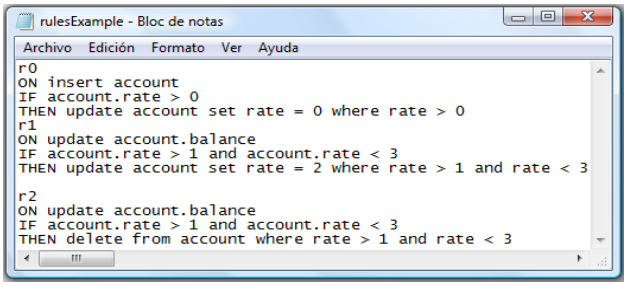


Figure 4. Rules of the CCPN in Figure 3

V. Conclusion

Our tool can be an aid to bring formal methods to non-experts in an easy way and it can improve the quality of active systems since it also performs formal evaluation on the rule base.

REFERENCES

[1] X. Li, J. Medina-Marín, and Chapa S., Applying Petri Nets on Active Database Systems, *IEEE Trans. on System, Man, and Cybernetics, Part C: App. and Rev.*, Vol. 37, No. 4, pp. 482-493, 2007.
[2] A.M. Ericsson, and M. Berndtsson, REX, the Rule and Event eXplorer, in *Proc. of the 2007 Intl. Conf. on Dist. Event-based systems*, Toronto, Ontario, Canada, vol. 233, pp. 71 - 74, 2007.
[3] S. Chakravarty, and S. Varkala, Dynamic Programming Environment for Active Rules, in *Proc. of the 7th Intl. Baltic Conference on Databases and Information Systems*, pp. 3 – 16, 2006.

From BPEL to SAL And Back: a Tool Demo on Back-Annotation with VIATRA2

Ábel Hegedüs, István Ráth and Dániel Varró
Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary
Email: {hegedusa,rath,varro}@mit.bme.hu

***Abstract*—Model-driven analysis aims at detecting design flaws early in high-level design models by automatically deriving mathematical models. These analysis models are subsequently investigated by formal verification and validation (V&V) tools, which may retrieve traces violating a certain requirement. Back-annotation aims at mapping back the results of V&V tools to the design model in order to highlight the real source of the fault, to ease making necessary amendments.**

In this tool demonstration we present an end-to-end V&V tool for BPEL business processes that includes complex back-annotation support for representing V&V results as process execution traces in the design environment.

***Keywords*-back-annotation; traceability modeling; dynamic traceability**

I. Overview

Model transformations are increasingly involved in various fields of software engineering, from business process modeling through formal verification to code generation. The models acting as source and target for the transformations often represent different domains, thus identification of correspondence between them is non-trivial. Although in the field of critical systems and services the precise recording of *traceability information* is a strict requirement, in most industrial environments only ad-hoc solutions are used for handling this information.

Throughout the lifecycle of a system or product traceability information is generated and used for various tasks. The correspondence records are most often created at the time when the target model is produced using the source model during the execution of the transformation. This information can be later used for validation, verification, change management, maintenance or back-annotation. It is important to note that the traceability information itself may be accessed with model transformations thus *model-based traceability solutions* are advantageous.

Model-driven analysis (illustrated in Figure 1) aims at revealing conceptual flaws early in the design process. In the typical approach, high-level design models (UML, BPEL [1], SysML, etc.) are automatically transformed into mathematical models (e.g. Petri nets, transition systems, process algebras) to carry out analysis by formal methods.

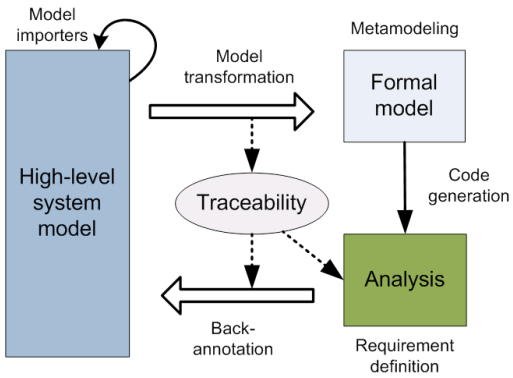


Figure 1. Methodological overview

The results of the analysis are then attempted to be back-annotated to the original source model to highlight flaws directly in the design models.

In case of dynamic modeling languages (e.g. statecharts, workflows), the back-end formal analysis tools frequently carry out simulation or model checking to ensure the functional correctness of the design using analysis models like Petri nets, process algebras or labeled transition systems. As a result, back-end analysis tools retrieve an execution trace (run) of the system as a designated or counter example.

A. Back-annotation

Counter-example traces can be very complex, resulting in well over 100 elementary steps in industrial scenarios. As a result, systems designers need to bridge a significant conceptual gap when they try to interpret what a counter-example means in the *original* design model. Back-annotation aims at automatically mapping back the results of V&V tools to the original design model in order to highlight the real source of the flaw.

Due to the semantic differences between high-level design models and lower-level formal analysis models, we argue in this paper that the *general back-annotation problem aiming to map a trace of a target model to a trace in the source model* can be very complicated. This is due

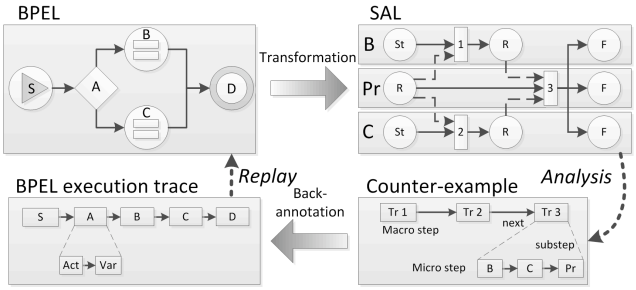


Figure 2. BPEL Verification overview

to the fact that most traditional source-to-target (design-to-analysis) model transformations carry out an abstraction, thus they are not reversible. However, in order to completely hide the underlying formal model from systems engineers, model-driven analysis need to provide automated support the back-annotation of target (analysis) models to the source (design) model.

Unfortunately, existing back-annotation approaches are either dedicated to the source and target languages, or they make strong assumptions on the back-annotation problem.

B. BPEL to SAL and back

In the demonstrated tool, BPEL business processes can be verified against requirements and the results are presented as an execution run of the business process. Since the BPEL standard does not define precise formal semantics for business process, formal languages (such as Petri nets or labeled transition systems) are used to formalize their behavior. In our tool, BPEL processes are transformed (using the VIATRA2 framework [2]) to the labeled transition system language of the Symbolic Analysis Laboratory (SAL) [3] model checking framework (illustrated in Figure 2). Then the analysis is carried out using SAL and results in a counter-example which contains a sequence of transitions which violate a requirement. The counter-example is back-annotated to a BPEL execution and can be replayed in the high-level domain.

We developed a model-based approach for persisting, handling and back-annotating execution traces (e.g. counter-examples, simulation traces). The foundation of the approach is a *generic metamodel for execution traces* which can describe the traces of various discrete event-based languages. On top of this generic metamodel, we defined *language-independent methods for navigating* through traces, and *replay* their effects on the model itself (i.e. the SAL transition system). Furthermore, the *back-annotation of traces* is possible by mapping the steps of the analysis trace to steps of the original trace [4].

Generic trace metamodel: While execution traces are traditionally interpreted as a sequence of elementary operations, in our approach, we use hierarchical trace models (see Figure 2) consisting of *micro steps* (atomic operations, m) and *macro steps* (complex operations, M), which is compliant with recent approaches [5] to define semantics for big-step DMLs like statecharts.

In the example illustrated in Figure 2, a macro step of the counter-example is firing a transition (e.g. 1) in the SAL system, while the micro steps are variable assignments (e.g. variable B changes from St to R). In the BPEL execution trace, a macro step is some activity event (e.g. B starts, A finishes), while micro steps are either activity state changes (Act) or variable manipulations (Var).

Trace replay and Visualization: The demonstrated tool is able to replay the persisted execution traces of both analysis and design models as long as they conform to the generic metamodel. Replaying can be navigated through a user interface component of the Eclipse framework where arbitrary BPEL processes and associated traces are handled. The tool also includes an intuitive graphical representation of execution trace replaying with a modified Eclipse BPEL Designer [6]. The activities and variables of the BPEL process are colored based on their runtime state.

C. Target audience amd benefits of using the tool

We believe that our tool demonstration can be beneficial to the following groups:

- Researchers and industry members with an interest in formal analysis and verification, as our tool includes both a V&V method and an intuitive approach to represent analysis results in the design language context.
- Users of the BPEL language and generally process modeling practitioners, who can benefit from a presentation on how processes can be verified design-time using hidden formal methods.
- Model transformation enthusiasts, who can gain an insight on the use of the VIATRA2 framework in a complex scenario including back-annotation and modeling of execution traces.

The benefits of using the presented tool is the ability to exploit the features of hidden formal methods for verifying business processes with precise formal models while being able to examine the results in the original design perspective.

II. TOOL DEMO CONTENTS

The tool demonstration is separated into three main parts. First a short preliminary part introduces the main languages and techniques used through the demonstration. Next a longer part deals with the presentation of the tool itself and its uses in a similar order to model-driven engineering methodology (see Figure 1). Finally, the underlying model transformation techniques are detailed using the VIATRA2 framework.

A. Preliminaries

1) **BPEL Overview**
   First, we give a brief introduction on the BPEL process description language and show the usage of the Eclipse BPEL Designer developer tool with creating a sample business process that we will use as an example in the rest of the tutorial.
2) **VIATRA2 basics**
   Here, we introduce the VIATRA2 model transformation framework including its model space, metamodeling capabilities, and transformation engine. We use the created BPEL process as an example to show how external models are imported into the model space as static models and how VIATRA2 is used for code generation to create external files.
3) **Model checking with SAL - a quick overview**
   Finally, we present the SAL model checking framework and its transition system description language. We show how linear temporal logic theorems are used to define requirements on the transition systems and are verified by the SAL model checker. Furthermore, we focus on the counter-examples that are the result of model checking and how they represent execution traces of the SAL model.

B. Applying hidden formal methods

1) **Verification Tool UI and features**
   First, we show our approach in creating a user interface for hidden formal methods. The created tool is embedded in Eclipse and incorporates all the features detailed above. BPEL processes can be transformed to SAL and common requirements can be defined without expertise in temporal logic. Requirements can be verified using the SAL model checker from the same tool and the results can be viewed as well.
2) **Customizing the modeling tool**
   The BPEL Designer tool provides only process definition support, it cannot show the dynamic state of a process instance. Here, we briefly outline how we modified the tool to be able to give graphical representation to the dynamic state of BPEL processes.
3) **Trace handling tool and features**
   Finally, we present the execution trace controller tool that uses the output of the SAL2BPEL transformation to update the dynamic state of the BPEL process instance in the customized BPEL Designer tool.

C. Underlying VIATRA2 technology highlights

1) **Metamodeling**
   First, we show how the operational semantics and the dynamic behavior of a language can be used to define metamodels in which trace information can be persisted during simulation execution or using a counter-example after model checking.
2) **Transformation development**
   In this part, we present the transformation language of VIATRA2 and its main features which are used to implement the trace replaying and back-annotation.
3) **Visualization**
   Finally, we show the visualization capabilities of the framework, that can be used for visualizing model and pattern graphs. Domain-specific graph layouts help in debugging the transformations during development, while they are also used for visualizing static and dynamic traceability models in order to aid verification and back-annotation [7].

III. TOOL AVAILABILITY AND MATURITY

The demonstrated tool is partially the result of the SENSORIA European project where it was also used as a demonstration tool [8] for the traceability visualization and back-annotation capabilities of the VIATRA2 framework. The tool itself is the result of a year of research and development, and is available as an Eclipse update site, since it is entirely developed as Eclipse components. Although the SAL model checker framework is not Eclipse-based and has to be installed and configured independently, it is a completely free tool available from the Symbolic Analysis Laboratory Website[1]. A simple webpage describing the tool and the installation procedure is available[2].

IV. SCREENSHOTS FROM THE TOOL

In this section we present the actual user interface of the BPEL Verification and Animation tool and the underlying VIATRA2 framework through screenshots.

Figure 3 shows the user interface of the BPEL Verification Tool, where the structural transformation (from BPEL to SAL) is executed and requirements can be defined, which are then verified against the business process. Generic requirements can be selected from a drop-down list thus relieving the user from using Linear Temporal Logic [9] expressions. Process-specific requirements can be captured using the LTL expression field, which also includes a basic syntax checker.

The tool provides an option for choosing either the symbolic or the bounded model checker of the SAL framework (which may have different performance on the same LTL expression). Finally, verification of a given requirement can be started with the *Check Property* button. Since model checking may take a long time for complex processes, the verification runs as a background task.

Figure 4 shows a part of a counter-example returned by the SAL model checker for a requirement that is violated by the business process. The LTL expression for the requirement (variable is never read while uninitialized) is highlighted in the upper part. The steps of the counter-example are presented in a textual format (retrieved straight

[1] http://sal.csl.sri.com/
[2] http://mit.bme.hu/~hegedusa/exectraces/

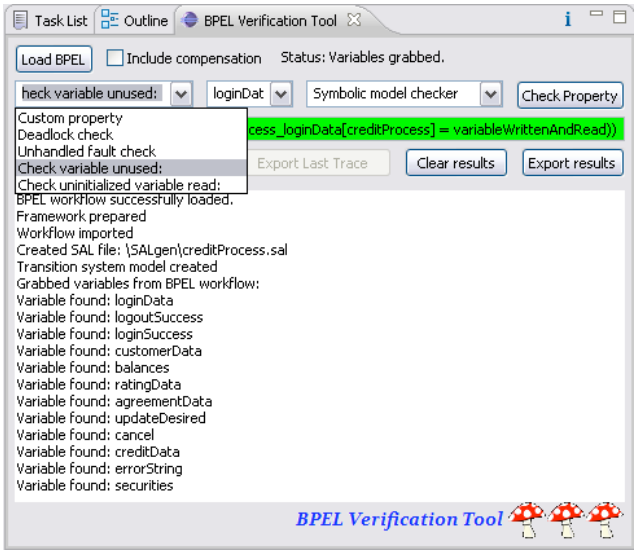


Figure 3. Aided business process requirement definition

from the SAL output) at this stage. Each step (e.g. *Step 90-91*) contains information about which transition fired (and where it is located in the transition system description) and the values of variables that changed as a result of the transition. The counter-example can be exported at any time into a file with a click of a button (*Export Last Trace*).

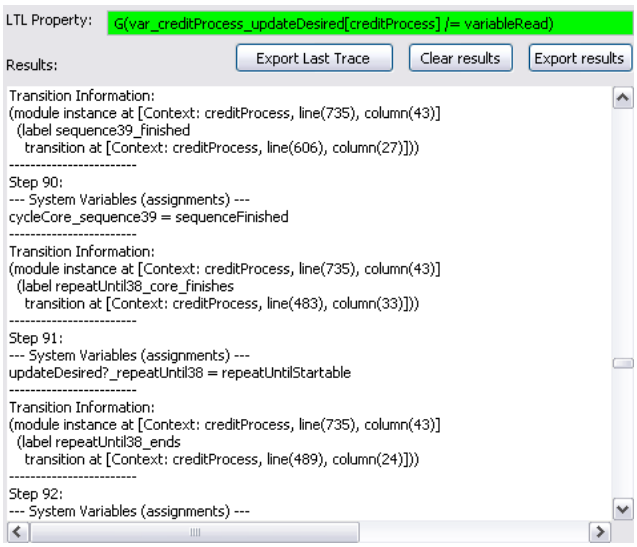


Figure 4. Verification results before back-annotation

Figure 5 shows the BPEL Animation Controller view, where exported counter-examples (traces) can be opened (*Load Trace*). After the textual file is processed, the VIATRA2 framework initializes the trace models and the back-annotation transformation. When the framework is ready, the navigation buttons can be used to animate the process execution. Apart from step-by-step navigation (*Step back/forward*), the tool also includes continuous animation mode (*Animate!/Stop*), quick return to the initial state (*Reset*) and animation speed-up (*Fast stepping*) for easier handling of long traces. Finally, the underlying model space can be saved for further use (*Save Modelspace*).

Figure 5. Animation controller

Figure 6 shows the customized BPEL Designer at a given state during the animation of an example BPEL process. The activities and variables of the process are colored depending on their dynamic state. Thus the counter-example of the model checker can be observed visually in the design perspective as an execution of the BPEL process. For the activities, light blue means *startable* state, light green *active*, dark green *finished*. For variables, yellow is *uninitialized* state, green is *correct* and red is *faulty*.

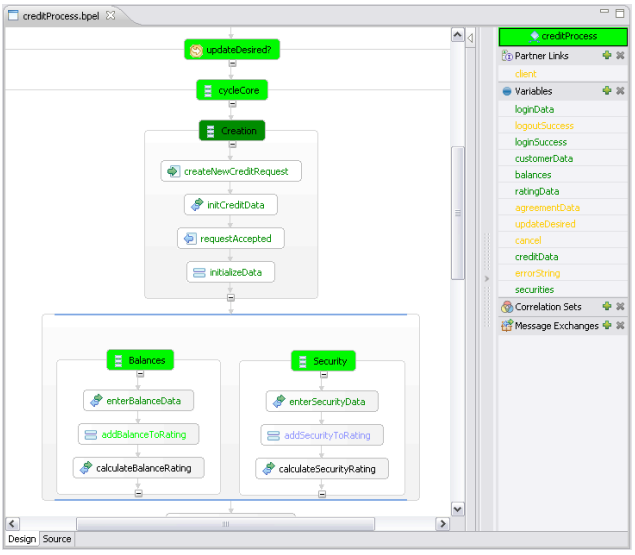


Figure 6. Animation of execution trace

Figure 7 shows the default model space editor of the VIATRA2 framework. Various models are stored in a containment hierarchy visualized in a tree view and model transformation programs can operate on the whole model space. Note that the models for the BPEL process and SAL systems are in different subtrees and the static, dynamic and trace models are separated as well. The metamodels which the various models conform to are also stored in the same model space.

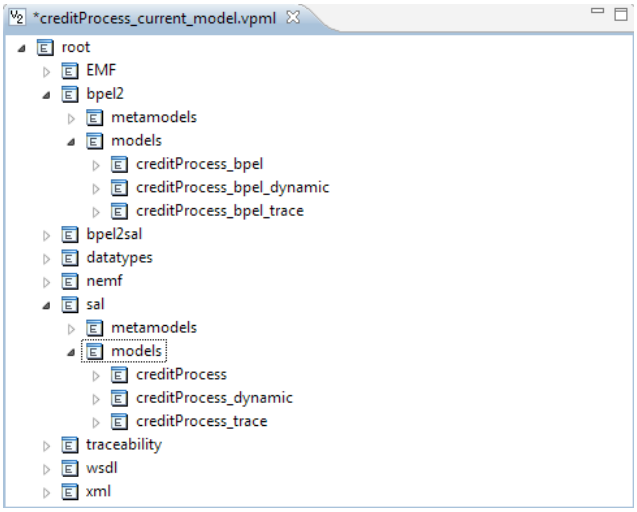


Figure 7. Modelspace view (BPEL and SAL models)

Figure 8 shows the static traceability model presented using a domain-specific layout through the model space visualization component of the VIATRA2 frameworkvisualization framework. A relevant subset of BPEL model elements are grouped on the left, the records of the static traceability model are placed in the middle, while corresponding SAL model elements are displayed on the right.

Figure 8. Visualized static traceability model (BPEL and SAL models)

Figure 9 shows the dynamic traceability model using the same visualization component, though with a different layout. The steps and substeps of the BPEL trace model are grouped on the left, while the steps of the SAL trace model, which are used in the back-annotation transformation are displayed on the right.

ACKNOWLEDGMENT

This work was partially supported by the EU project SecureChange (ICT-FET-231101) and CERTIMOT (ERC_HU_09).

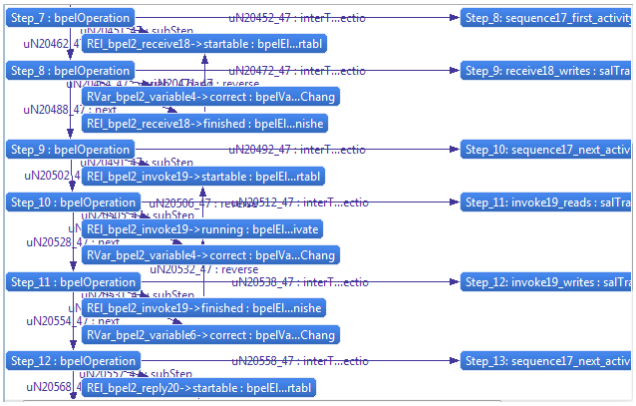


Figure 9. Visualized dynamic traceability model (BPEL and SAL traces)

REFERENCES

[1] OASIS, "Web Services Business Process Execution Language Version 2.0 (OASIS Standard)," 2007, "http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html".

[2] Fault Tolerant System Research Group, BME, "VIATRA2 Model Transformation Framework, An Eclipse GMT Subproject," http://www.eclipse.org/gmt/VIATRA2/.

[3] S. Bensalem, V. Ganesh, Y. Lakhnech, C. M. noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari, "An overview of SAL," in *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, C. M. Holloway, Ed., Hampton, VA, jun 2000, pp. 187–196.

[4] Á. Hegedüs, I. Ráth, and D. Varró, "Back-annotation of Simulation Traces with Change-Driven Model Transformations," in *Proceedings of the Eigth International Conference on Software Engineering and Formal Methods*, 2010, accepted. http://home.mit.bme.hu/~hegedusa/assets/publ/sefm10-back-ann.pdf.

[5] S. Esmaeilsabzali and N. A. Day, "Prescriptive semantics for big-step modelling languages," in *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Proceedings*, ser. LNCS, D. S. Rosenblum and G. Taentzer, Eds., vol. 6013. Springer, 2010, pp. 158–172.

[6] "Eclipse BPEL Designer, An Eclipse Project," http://www.eclipse.org/bpel/.

[7] Á. Hegedüs, Z. Ujhelyi, I. Ráth, and Á. Horváth, "Visualization of Traceability Models with Domain-specific Layouting," in *Proceedings of the Fourth International Workshop on Graph-Based Tools*, 2010, accepted.

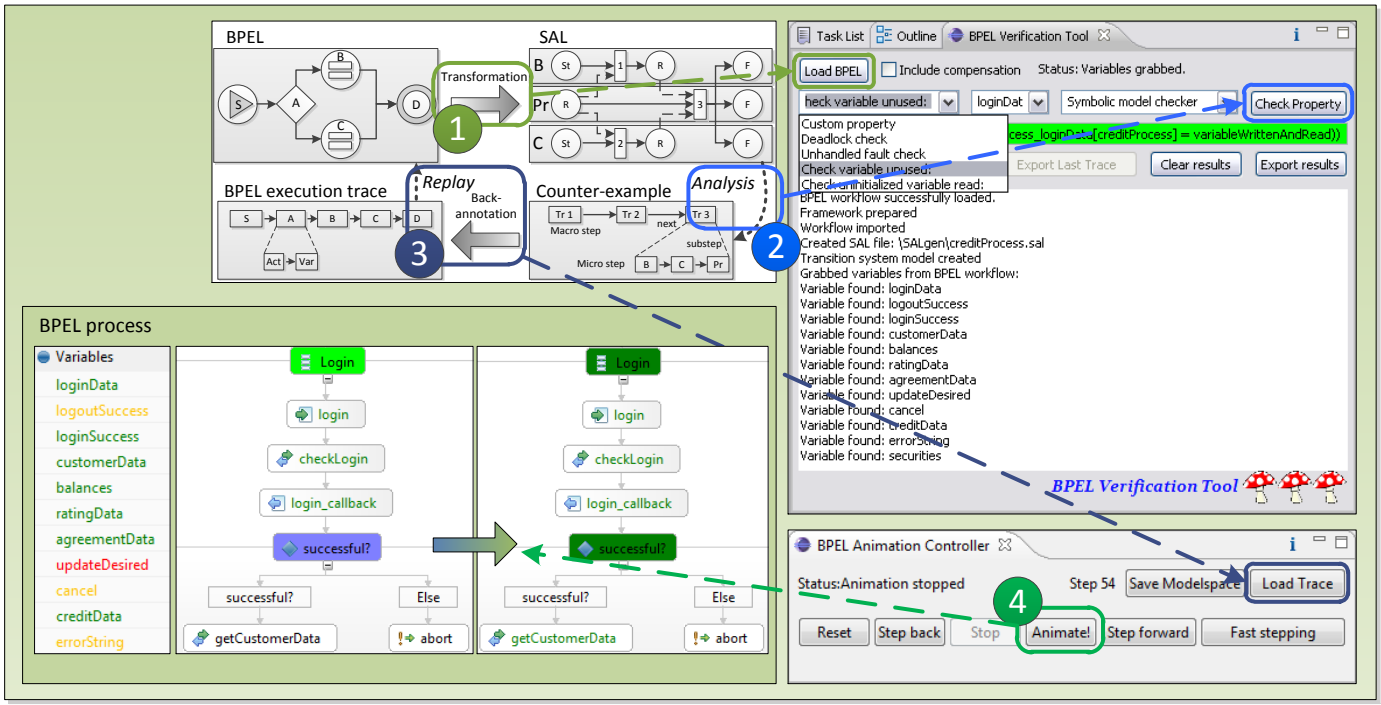
[8] L. Gönczy, Á. Hegedüs, and D. Varró, "Methodologies for Model-Driven Development and Deployment: an Overview," in *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA project on Software Engineering for Service-Oriented Computing*, M. Wirsing, Ed. Springer-Verlag, 2010, to appear.

[9] E. A. Emerson, *Temporal and Modal Logic*. Elsevier, 1990, vol. B, Formal Models and Semantics, pp. 995–1072.

BPEL to SAL and Back: An End-to End Business Process Verification Tool

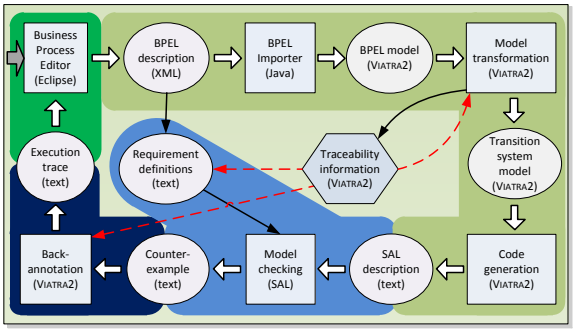
Ábel Hegedüs, István Ráth, and Dániel Varró (Budapest University of Technology and Economics)
{hegedusa,rath,varro}@mit.bme.hu

Business processes orchestrated from web-services using the Business Process Execution Language standard (BPEL) are widely used in business-critical and high-availability systems. In order to detect design flaws early, various approaches exist for *transforming BPEL processes* into mathematical models on which *analysis* (verification & validation) can be carried out, e.g. with the Symbolic Analysis Laboratory (SAL) framework. The results of the analysis (often a sequence of steps, i.e. counter-example or *trace*) are *back-annotated* into the original BPEL process, where they can be *replayed* for the developer. By providing automated solutions for these steps, *business process verification is possible using hidden formal methods*.



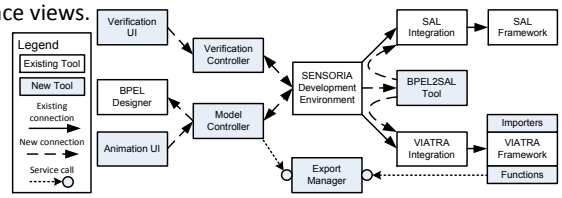
Methodological overview

The initial input of the tool is a *BPEL process description* (stored as an XML file), often created using a graphical editor, which is imported into the model space of the VIATRA2 model transformation framework using a Java-based *importer*. The resulting *BPEL model* is *transformed* into a *transition system model*, then the *SAL description* is created from this model using *code generation*. The *model checking* capabilities of the SAL framework is used to verify the BPEL process against *requirements* (defined in Linear Temporal Logic). The results (*counter-example*) are imported into VIATRA2 and transformed into BPEL trace (*back-annotation*). Traceability information created during the transformation is stored as a model and is used repeatedly throughout the method.



Integrated toolchain

The complete verification and animation tool is built using the Eclipse framework by integrating existing techniques and creating new components and links between the parts of the toolchain. The components on the left side of the figure provide the user interface and are connected with two controllers. The controllers handle the low-level transformation and verification tools in order to prepare data for the user interface views.



Tool features

- Transform BPEL business processes into a precise and formal SAL transition system.
- Guided definition of generic and process-specific requirements.
- Automated model checking of defined require-ments using the SAL framework.
- Model-based persistence and handling of SAL and BPEL simulation traces.
- Automated back-annotation of SAL simulation traces into BPEL execution traces.
- User-guided, graphical animation of BPEL execution traces based on SAL traces.
- Graph visualization of static and dynamic traceability models using domain-specific layout algorithms.

Publications & Links

- L. Gönczy, Á. Hegedüs, and D. Varró, "**Methodologies for Model-Driven Development and Deployment: an Overview**," in *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA project on Software Engineering for Service-Oriented Computing*, M. Wirsing, Ed. Springer-Verlag, 2010
- Á. Hegedüs, I. Ráth, and D. Varró, "**Back-annotation of Simulation Traces with Change-Driven Model Transformations**," in *Proceedings of the 8th International Conference on Software Engineering and Formal Methods*, 2010
- Á. Hegedüs, Z. Ujhelyi, I. Ráth, and Á. Horváth, „**Visualization of Traceability Models with Domain-specific Layouting**", in *Proceedings of the 4th International Workshop on Graph Based Tools*, 2010

- Tool available at: http://mit.bme.hu/~hegedusa/exectraces/
- VIATRA2: An Eclipse GMT Subproject: http://www.eclipse.org/gmt/VIATRA2/
- Symbolic Analysis Laboratory: http://sal.csl.sri.com
- Eclipse BPEL Project: http://www.eclipse.org/bpel/

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group
H-1117 Budapest, XI. Magyar Tudósok krt. 2
Phone: (+36 1) 463 3579

MŰEGYETEM 1782

SEFM'2010
13-16th September, Pisa

B*lite*C: a tool for developing WS-BPEL applications

Luca Cesari, Rosario Pugliese and Francesco Tiezzi
Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
cesari.luca@gmail.com, rosario.pugliese@unifi.it, tiezzi@dsi.unifi.it

***Abstract*—WS-BPEL is imposing itself as a standard for orchestration of web services. However, there are still some well-known difficulties that make programming in WS-BPEL a tricky task. We present here B*lite*C, a software tool we have developed for supporting a rapid and easy development of WS-BPEL applications. B*lite*C translates service orchestrations written in B*lite*, a formal language inspired to but simpler than WS-BPEL, into readily executable WS-BPEL programs. We illustrate our approach by means of a practical programming example.**

***Keywords*-SOC; Web services; Compilers**

I. Introduction

In recent years, there has been an ever increasing acceptance of WS-BPEL [1] as a standard language for orchestration of *web services*, one of the most successful and well-developed implementations of the *Service-Oriented Computing* (SOC) paradigm. However, designing and developing WS-BPEL applications is a difficult and error-prone task. The language has an XML syntax which makes awkward writing WS-BPEL code by using standard editors. Therefore, many companies (among which e.g. Oracle and Active Endpoints) have equipped their WS-BPEL engines with graphical designers. Such tools are certainly suitable to develop simple business processes, but turn out to be cumbersome and ineffective when dealing with more complex applications. Further difficulties derive from the fact that WS-BPEL is equipped with such intricate features as concurrency, multiple service instances, message correlation, long-running business transactions, termination and compensation handlers. Most of all, WS-BPEL comes without a formal semantics and its specification document, written in 'natural' language, contains a fair number of acknowledged ambiguous features that may give rise to different interpretations. These ambiguities have led to engines implementing different semantics (see [2]) and, hence, have undermined portability of WS-BPEL programs across different platforms. Finally, portability is further compromised since the deployment procedure of WS-BPEL programs is not standardised. In fact, to execute a WS-BPEL program, besides the associated WSDL [3] document that describes the program's public interfaces, different engines require different (and not integrable) *process deployment descriptors*, i.e. sets of configuration files that describe how the program should be deployed.

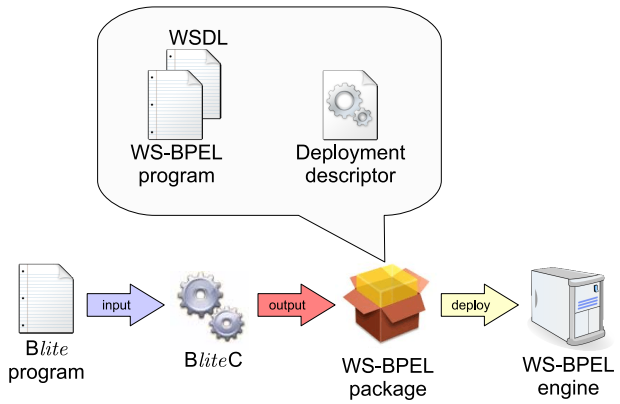


Figure 1. B*lite*C workflow

To overcome these difficulties, we have developed B*lite*C, a software tool that accepts as an input a specification written in the lightweight orchestration language B*lite* [2] and returns the corresponding WS-BPEL program together with the associated WSDL and deployment descriptor files.

B*lite* is closely inspired to WS-BPEL. It is the result of a tension between handiness and expressiveness. While the set of WS-BPEL constructs is not intended to be a minimal one, to keep the language manageable, the design of B*lite* only retains the core features of WS-BPEL. It follows that B*lite* is simpler and more compact than WS-BPEL, although it maintains the same descriptive power. Using B*lite* for initially specifying a service orchestration offers some significant advantages. From the one hand, B*lite* textual notation is certainly more manageable than those, possibly graphical, notations proposed for WS-BPEL. From the other hand, B*lite* is equipped with a formal operational semantics that clarifies all ambiguous and intricate aspects of WS-BPEL.

B*lite*C further simplifies the programmers work by automatizing the deployment procedure. In fact, the returned files are properly packaged to be immediately executable in a WS-BPEL engine. Currently, these packages are intended to be deployed on ActiveBPEL [4] that, according to [2], is one of the freely available WS-BPEL engines that better complies with the WS-BPEL specification. The workflow of use of B*lite*C is graphically depicted in Figure 1.

In Section II we shall provide some insights into the architecture of B*lite*C, while in Section III we illustrate the functioning of B*lite*C through an example of a virtual credit

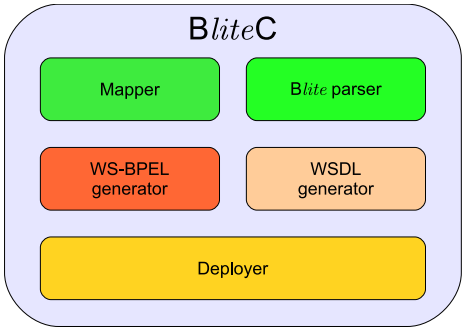


Figure 2. BliteC architecture

card service. We refer the interested reader to [5] for a more complete account of BliteC.

II. BliteC ARCHITECTURE

BliteC[1] is developed in Java[2] to guarantee its portability across different platforms, to exploit the well-established libraries for generating parsers and for manipulating XML documents, and because Java is the reference language for the applications designed around WS-BPEL. Besides the standard Java libraries, we have used JDOM [6] for creating and managing XML documents, JavaCC [7] for generating the parsers that validate the input documents, and JJTree[3] for allowing the parsers to build parse trees (already arranged to support the Visitor design pattern [8]).

The architecture of BliteC is graphically depicted in Figure 2. The tool is composed of five main components:

- *Mapper* parses the declarative part of the input Blite program (containing details necessary for the deployment) and initializes a map that associates each declared object (e.g. partner link, literal, variable, ...) to its name;
- Blite *parser* analyzes the Blite specification within the input program, completes the map created by Mapper and creates the parse tree of the Blite specification;
- WS-BPEL and WSDL *generators* use the data produced by the above components to generate a WS-BPEL process and the associated WSDL document;
- *Deployer* generates the deployment descriptor and packages all created documents into a deployable file.

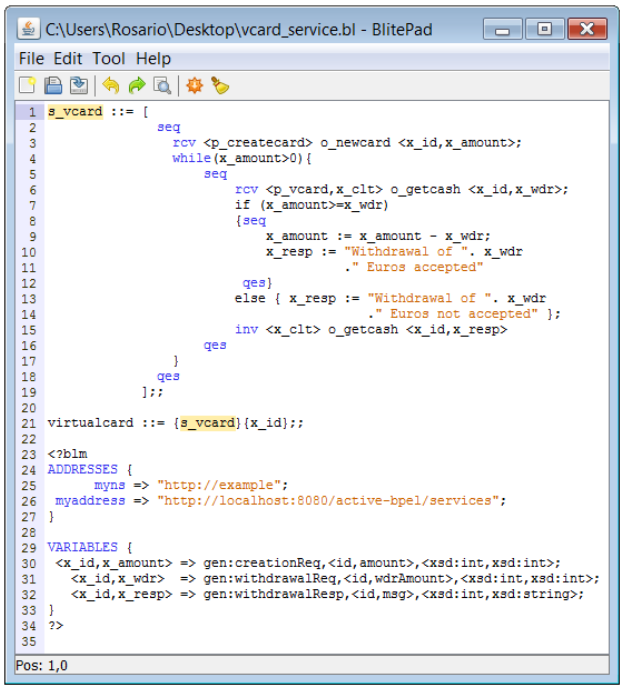
Any text editor can be used to write Blite programs. Anyway, to simplify the task, we provide users with a customized version of jEdit[4] [9] equipped with specific features supporting programming in Blite, such as syntax highlighting, auto indentation and direct compiling. The files for the customization can be downloaded with the

[1]BliteC is a free software; it can be downloaded from http://rap.dsi.unifi.it/blite and redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation.

[2]JRE and JDK version 6.

[3]JJTree is included within JavaCC.

[4]jEdit is a programmer's text editor written in Java released as free software with full source code, provided under the terms of the GPL 2.0.




```
s_vcard ::= [
              seq
                rcv <p_createcard> o_newcard <x_id,x_amount>;
                while(x_amount>0){
                    seq
                        rcv <p_vcard,x_clt> o_getcash <x_id,x_wdr>;
                        if (x_amount>=x_wdr)
                        {seq
                            x_amount := x_amount - x_wdr;
                            x_resp := "Withdrawal of ". x_wdr
                                      ." Euros accepted"
                         qes}
                        else { x_resp := "Withdrawal of ". x_wdr
                                         ." Euros not accepted" };
                        inv <x_clt> o_getcash <x_id,x_resp>
                    qes
                }
              qes
            ];;

virtualcard ::= {s_vcard}{x_id};;

<?blm
ADDRESSES {
      myns => "http://example";
 myaddress => "http://localhost:8080/active-bpel/services";
}

VARIABLES {
 <x_id,x_amount> => gen:creationReq,<id,amount>,<xsd:int,xsd:int>;
   <x_id,x_wdr>  => gen:withdrawalReq,<id,wdrAmount>,<xsd:int,xsd:int>;
   <x_id,x_resp> => gen:withdrawalResp,<id,msg>,<xsd:int,xsd:string>;
}
?>
```

Figure 3. BliteC development environment

BliteC distribution archive. The main advantage of jEdit with respect to more professional development environments is that it is multi-platform and lightweight. We are also implementing a development environment with similar features written in Java. Figure 3 shows a screenshot of our environment. In addition to the functionalities of the customized version of jEdit, our dedicated environment also provides text auto-completion, highlight of search results, local deploy and undeploy.

III. BliteC AT WORK ON A CREDIT CARD SERVICE

A virtual credit card is a prepaid non-physical credit card devised for safe online shopping. A Blite specification for creating and handling a virtual credit card is the following:

```
s_vcard ::=
  [ seq
      rcv <p_createcard> o_newcard <x_id,x_amount>;
      while(x_amount>0){
       seq
        rcv <p_vcard,x_clt> o_getcash <x_id,x_wdr>;
        if (x_amount>=x_wdr)
           { seq
              x_amount := x_amount - x_wdr;
              x_resp := "Withdrawal of ". x_wdr .
                        " Euros accepted"
           qes }
        else { x_resp := "Withdrawal of ". x_wdr .
                         " Euros not accepted"};
        inv <x_clt> o_getcash <x_id,x_resp>
       qes }
    qes ];;

virtualcard ::= {s_vcard}{x_id};;
```

The above specification defines a *deployment* `virtualcard` of the form `{s_vcard}{x_id}`, which associates the *corre-*

lation variable `x_id` (storing the card identifier) to the *service* `s_vcard`. A service provides a 'top-level' scope, denoted by square brackets enclosing a primary activity that, in this case, is a sequence of activities of the form `seq a_1 ; ... ; a_n qes`.

A new card is created by invoking the operation `o_newcard`, provided by means of the *receive* activity `rcv <p_createcard> o_newcard <x_id,x_amount>`, and specifying a card identifier and the initial amount. The *partner link* `<p_createcard>` is used here for defining a one-way interaction. The created instance allows the card holder to perform withdrawals by repeatedly invoking the request-response operation `o_getcash`, provided by means of `rcv <p_vcard,x_clt> o_getcash <x_id,x_wdr>`, until the card is empty. Here, differently from the initial receive, the partner link indicates two partners to define a request-response interaction. For each withdrawal request, the money availability is checked and a message, stored in `x_resp` by means of an *assignment*, is sent back by the *invoke* activity `inv <x_clt> o_getcash <x_id,x_resp>`. The fact that the invoke activity used for the reply is performed along the same operation of the second receive indicates that the two activities form a *synchronous* request-response interaction, hence the invoke will be translated into a WS-BPEL `<reply>` activity.

To properly translate a B*lite* specification into a deployable and executable WS-BPEL program, some configuration data (e.g. types, addresses, bindings, ...) have to be provided. B*lite*C requires the user to insert only the strictly necessary information. Such declarations must be included within `<?blm` and `?>`, and can occur in any position within a B*lite* program. In particular, in our example, since the service does not need to invoke other services, only its address and variables are explicitly declared:

```
<?blm
 ADDRESSES {
    myns => "http://virtualcard";
    myaddress => "http://XXX:8080/active-bpel/services";
 }
 VARIABLES {
   <x_id,x_amount> => gen:creationReq,<id,amount>,
                      <xsd:int,xsd:int>;
   <x_id,x_wdr>    => gen:withdrawalReq,<id,wdrAmount>,
                      <xsd:int,xsd:int>;
   <x_id,x_resp>   => gen:withdrawalResp,<id,msg>,
                      <xsd:int,xsd:string>;
 }
?>
```

Within the `ADDRESSES` block are specified the base for the namespaces used inside the generated files and the base for the address where the new service will be hosted (where `XXX` is the server's address where the ActiveBPEL engine is running). Instead, within the `VARIABLES` block, for example, the tuple of variables `<x_id,x_amount>` is typed as `gen:creationReq`, a message type that defines messages composed of two integer parts, `id` and `amount`. The namespace prefix `gen` indicates that the type will be generated by B*lite*C.

In the declarative part of a B*lite* program can be also inserted an `IMPORTS` block, to specify the addresses of the documents to be imported (typically, WSDL and XSD files), a `LITERALS` block, to specify literals (i.e. constant values), and a `PARTNERLINKS` block, to specify the type of the partner links used by the process (except for those used by the process to interact with its clients, which are automatically generated and typed by B*lite*C).

To compile the B*lite* program previously presented, we have to save the code into a file (named, e.g., `vcard_service.bl`) and execute the command

```
java -jar blite.jar vcard_service.bl
```

This way, the file `virtualcardProcess.bpr`, which is a WS-BPEL package directly deployable into ActiveBPEL, is generated. To deploy this file, it is sufficient to move it into the engine's deployment directory `bpr`. The same actions can be also performed by using the development environment mentioned in Section II.

The WS-BPEL file included in the generated package, where irrelevant details have been omitted, is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="virtualcardProcess" ... >
  <import location="virtualcard.wsdl" ... />
  <partnerLinks>
    <partnerLink name="cltPL" partnerLinkType="mwl:cltPLT"
                 myRole="p_vcard" />
    <partnerLink name="p_createcardPL"
                 partnerLinkType="mwl:p_createcardPLT"
                 myRole="p_createcard" />
  </partnerLinks>
  <variables>
    <variable name="var1"
              messageType="mwl:withdrawalReq" />
    <variable name="var2"
              messageType="mwl:withdrawalResp" />
    <variable name="var0"
              messageType="mwl:creationReq" />
  </variables>
  <correlationSets>
    <correlationSet name="x_idCorr"
                    properties="mwl:x_idProp" />
  </correlationSets>
  <faultHandlers>
    <catchAll>
      <sequence><compensate /><empty /></sequence>
    </catchAll>
  </faultHandlers>
  <sequence>
    <sequence>
      <receive partnerLink="p_createcardPL"
               operation="o_newcard"
               variable="var0"
               createInstance="yes">
        <correlations>
          <correlation set="x_idCorr" initiate="yes" />
        </correlations>
      </receive>
      <assign>
        <copy>
          <from variable="var0" part="id" />
          <to variable="var1" part="id" />
        </copy>
        <copy>
          <from variable="var0" part="id" />
          <to variable="var2" part="id" />
        </copy>
      </assign>
    </sequence>
```

```
<while>
  <condition>$var0.amount &gt; 0</condition>
  <sequence>
    <receive partnerLink="cltPL"
             operation="o_getcash"
             variable="var1">
      <correlations>
        <correlation set="x_idCorr" initiate="no" />
      </correlations>
    </receive>
    <if>
      <condition>
        $var0.amount &gt;= $var1.wdrAmount
      </condition>
      <sequence>
        <assign>
          <copy>
            <from>
              $var0.amount - $var1.wdrAmount
            </from>
            <to variable="var0" part="amount" />
          </copy>
        </assign>
        <assign>
          <copy>
            <from>
              concat('Withdrawal of ',
                     string($var1.wdrAmount),
                     ' Euros accepted')
            </from>
            <to variable="var2" part="msg" />
          </copy>
        </assign>
      </sequence>
      <else>
        <assign>
          <copy>
            <from>
              concat('Withdrawal of ',
                     string($var1.wdrAmount),
                     ' Euros not accepted')
            </from>
            <to variable="var2" part="msg" />
          </copy>
        </assign>
      </else>
    </if>
    <reply operation="o_getcash"
           partnerLink="cltPL"
           variable="var2">
      <correlations>
        <correlation set="x_idCorr" initiate="no" />
      </correlations>
    </reply>
  </sequence>
</while>
</sequence>
</process>
```

As expected, the resulting WS-BPEL code is more verbose and intricate than the B*lite* one. However, the performed translation turns out to be quite 'clean', in the sense that each B*lite*C activity has been translated into the corresponding WS-BPEL one without introducing 'junk' code.

Then, to check that the deploy succeeded, we can use the ActiveBPEL's administration console that can be accessed by using any browser at the address `http://XXX:8080/BpelAdmin`. By selecting `Deployed Processes` from the menu on the left-hand side, we obtain the list of the deployed processes among which `virtualcardProcess` should appear. Now, by selecting `Deployed services`, we can retrieve the URLs of the two WSDL files corresponding to the partner links for interacting with the service.

To test the service behaviour, we can use a tool for automatic generation of web service requests, as e.g. soapUI [10], and invoke the service by sending two SOAP messages: the first message creates a credit card identified by `1234` with `100` Euros as initial amount, while the second is a request for withdrawing `50` Euros from the same credit card. In response to the second message we get the string `Withdrawal of 50 Euros accepted` and, by selecting `Active Processes` from the console menu, we can verify that the card instance is still running. If we resend the withdrawal request we obtain the same response, but the instance status changes to `Completed`.

IV. Concluding remarks

We have presented B*lite*C, a software tool for supporting a rapid and easy development of WS-BPEL applications. The tool aims at solving some well-known programming problems of WS-BPEL caused by its XML syntax, lack of a formal semantics, and non-standardization of the deployment procedure.

Currently, WS-BPEL packages generated by B*lite*C are intended to be deployed on ActiveBPEL. This is just to demonstrate feasibility of our approach. In fact, B*lite*C has been designed so that the generation of deployment descriptors for different engines can be easily integrated, and we plan to enable it to produce packages also for other freely available engines, namely Oracle BPEL Process Manager [11], Apache ODE [12] and Beepell [13].

We also plan to enrich the B*lite*C development environment presented in Section II with further functionalities, such as deployment/undeployment facilities over remote servers, and debugging tools, such as automatic generation of web interfaces for invoking the created services and log recording based on an embedded dedicated web service.

For what concern the language B*lite*, we also intend to investigate its extension with some WS-BPEL constructs that at the time being have been left out, such as timed activities, event and termination handlers, and more sophisticated forms of fault and compensation handling involving named faults and compensation of specified scopes. We do not envisage any major issue in translating such constructs in WS-BPEL code, while their addition to B*lite* would require to significantly revise the formal definition of the operational semantics of the language.

Finally, we intend to develop formal analysis techniques, e.g. based on model checking, for B*lite* specifications. This way, we would be able to specify in B*lite* an orchestration scenario, validate its behaviour by using formal tools, and deploy it as a set of WS-BPEL programs.

References

[1] OASIS WSBPEL TC, "Web Services Business Process Execution Language Version 2.0." OASIS, Tech. Rep., April 2007.

[2] A. Lapadula, R. Pugliese, and F. Tiezzi, "A formal account of WS-BPEL," in *COORDINATION*, ser. LNCS, vol. 5052. Springer, 2008, pp. 199–215.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," W3C, Tech. Rep., 2001.

[4] "ActiveBPEL 5.0.2," October 2009, http://sourceforge.net/projects/activebpel502/.

[5] L. Cesari, A. Lapadula, R. Pugliese, and F. Tiezzi, "A tool for rapid development of WS-BPEL applications," in *SAC, SOAP track*. ACM, 2010, pp. 2438–2442, A full version of the paper will appear in the *ACM SIGAPP Applied Computing Review* and an extended technical report is available online at http://rap.dsi.unifi.it/blite/.

[6] "JDOM 1.1," April 2009, http://www.jdom.org.

[7] "JavaCC 4.2," April 2009, https://javacc.dev.java.net.

[8] G. Erich, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] "jEdit Programmer's Text Editor 4.3," http://www.jedit.org/.

[10] "soapUI 3.5," March 2010, http://www.soapui.org.

[11] "Oracle BPEL Process Manager 10.1.3," December 2007, http://www.oracle.com/technology/bpel.

[12] "Apache ODE 1.3.3," August 2009, http://ode.apache.org.

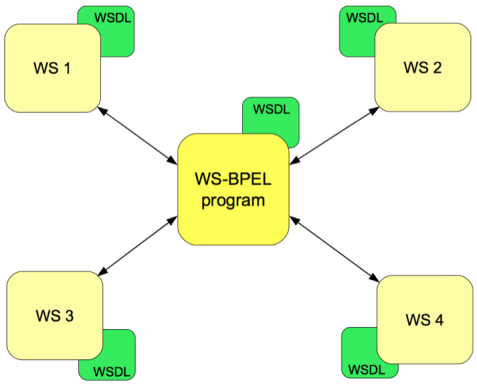
[13] T. Hallwyl, F. Henglein, and T. Hildebrandt, "A standard-driven implementation of WS-BPEL 2.0," in *SAC, SOAP track*. ACM, 2010, pp. 2472–2476.

BliteC: A Tool for Developing WS-BPEL applications

Luca Cesari, Rosario Pugliese and Francesco Tiezzi
Università degli Studi di Firenze, Dipartimento di Sistemi e Informatica
cesari.luca@gmail.com, rosario.pugliese@unifi.it, tiezzi@dsi.unifi.it

Service-Oriented Computing

- Web Services (WSs) are the most successful instantiation of the Service-Oriented Computing paradigm.
- A WS is basically a set of operations that can be invoked through the Web via XML messages complying with given standard formats.
- The OASIS standard WS-BPEL [1] is the most widespread language for programming WSs composition (the so-called WSs *orchestration*).
- A WS-BPEL program is a WS in its own right; this enables a compositional approach.



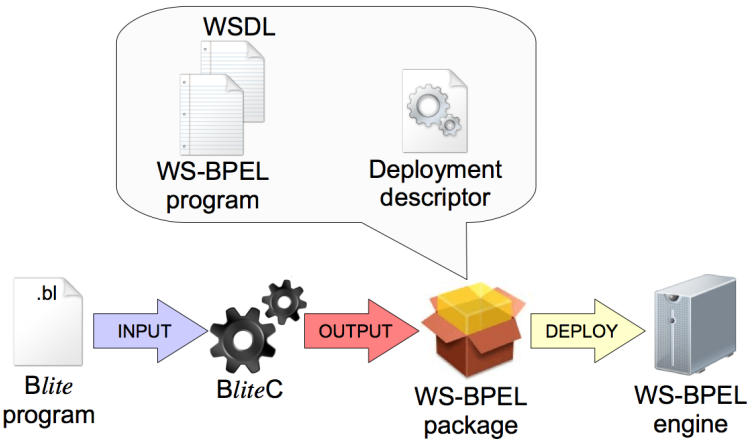
Motivations

Developing WS-BPEL applications is a difficult and error-prone task:

- WS-BPEL has an XML syntax (hard to read and write by human);
- WS-BPEL is equipped with intricate features: concurrency, multiple instances, message correlation, long-running transactions, ...;
- WS-BPEL comes without a formal semantics (its specification document is written in natural language); this has led to engines sometimes behaving differently [2];
- Deployment of WS-BPEL programs is not standardised.

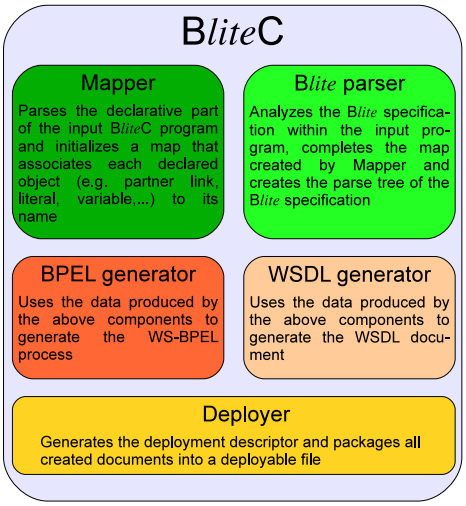
Our Proposal: the BliteC Tool

- BliteC [4] is a software tool for supporting a rapid and easy development of WS-BPEL applications.
- BliteC accepts as an input a service orchestration written in Blite [2] (i.e. a lightweight formal language inspired to but simpler than WS-BPEL) and returns the corresponding readily executable WS-BPEL program (i.e. a package containing the WS-BPEL program together with the associated WSDL document and deployment descriptor file).
- Currently, the generated WS-BPEL packages are intended to be deployed in ActiveBPEL [3].
- BliteC is a free, open-source software developed in Java; it can be downloaded from `http://rap.dsi.unifi.it/blite` and redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation.



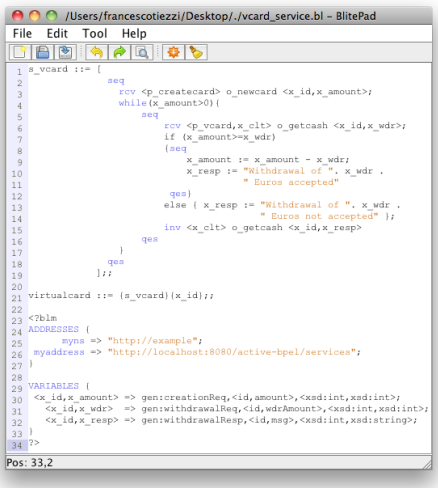
BliteC's Architecture

BliteC is composed of five main components.



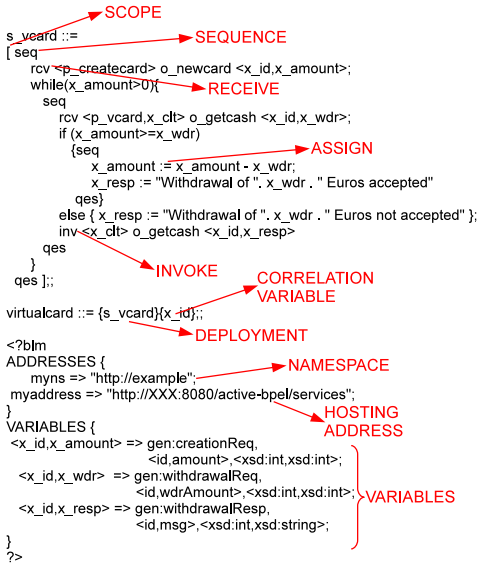
BliteC's IDE

Blite programs can be written by using any text editor or the dedicated development environment.



BliteC at Work

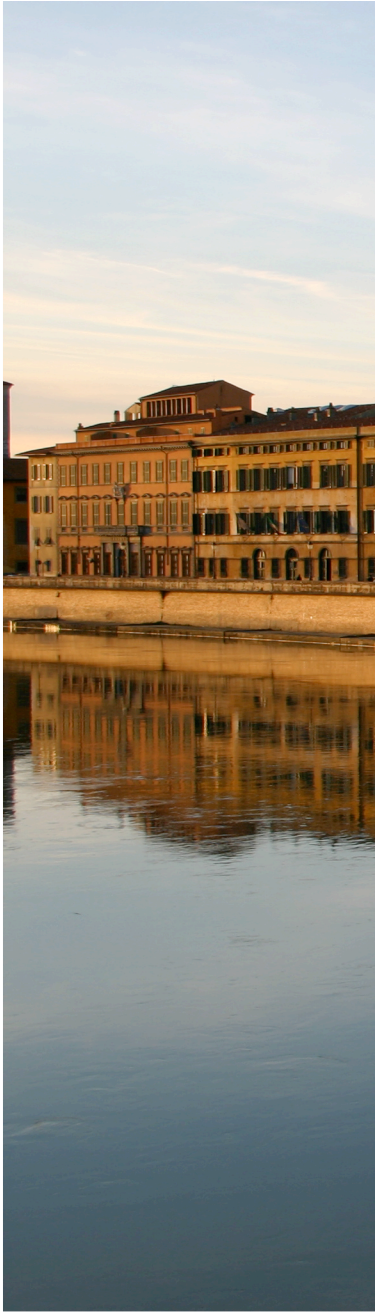
- A Blite program for creating and handling a virtual credit card.



- To compile the program, save the code into a file, e.g. `vcard.bl`, and execute the command `java -jar blite.jar vcard.bl`
- To deploy the generated file, i.e. the WS-BPEL package corresponding to the program, move it into the engine's deployment directory.

References

[1] OASIS WSBPEL TC. Web Services Business Process Execution Language v2.0. Technical report, OASIS, April 2007.
[2] A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. In *COORDINATION*, volume 5052 of *LNCS*, pages 199–215. Springer, 2008.
[3] ActiveBPEL 5.0.2, October 2009. `http://sourceforge.net/projects/activebpel02/`.
[4] L. Cesari, A. Lapadula, R. Pugliese, and F. Tiezzi. A tool for rapid development of WS-BPEL applications. In *SAC, SOAP track*, pages 2438–2442. ACM, 2010. An extended version will appear in the *ACM SIGAPP Applied Computing Review*.



ISTITUTO DI SCIENZA E TECNOLOGIE
DELL'INFORMAZIONE "A. FAEDO"

Published By
Consiglio Nazionale delle Ricerche
Area della Ricerca CNR di Pisa
Via Moruzzi, 1 - 56124 Pisa, Italy

ISBN 978-88-7958-006-9

ISTI Editorial 2010-ED-003