Integrating SMT-Solvers in Z and B Tools

Conference Paper · February 2010

DOI: 10.1007/978-3-642-11811-1_45 · Source: DBLP

4 authors, including:

Alessandro Gurgel
Pontifícia Universidade Católica do Rio de Janeiro
11 PUBLICATIONS **79** CITATIONS
SEE PROFILE

Marcel V M Oliveira
Universidade Federal do Rio Grande do Norte
64 PUBLICATIONS **634** CITATIONS
SEE PROFILE

David Déharbe
ClearSy System Engineering
114 PUBLICATIONS **940** CITATIONS
SEE PROFILE

ABZ 2010
Orford, Québec, Canada
February 22-25, 2010

Short Papers

Table of Contents

A Proof based approach for formal verification of transactional BPEL web services

Idir AIT SADOUNE and Yamine AIT AMEUR

LISI/ENSMA-UP
Tlport 2 -1, avenue Clment Ader, BP 40109, 86961, Futuroscope-Poitiers, France
{idir.aitsadoune,yamine}@ensma.fr

Abstract. In recent years, Web services are used in various operations that access and manipulate critical resources such as databases. To maintain these resources in a coherent state, the use of the adequate mechanisms is necessary and appropriate description languages provides just the opportunity to describe the behavior, to handle faults and to compensate activities, but does not ensure that the execution context remains consistent. This paper describes an undergoing work where we propose a formal approach to isolate transactional parts of a composed Web Service which are controlled later by fault and compensation mechanisms.

1 Introduction

The Service-Oriented Architectures (SOA) are increasingly used in various application domains. Today we find various Services that operate on the Web and access various critical resources such as databases. Some of these services performing database transactions are called transactional web services. This kind of Services must verify the relevant constraints related to transactional systems. In our work, we focus on web services described with BPEL [1].

In the BPEL[1] language, a composite Web Service is implemented by a process that consists of activities such as the messaging activities *invoke* and *reply*, which are used for interacting with the other web services and the structured activities *sequence*, *flow* and *scope*, which act as containers for their nested activities. BPEL provides some support for transactions through its *fault* and *compensation* handlers, which allow undoing the effects of completed activities.

In most related work [2, 3], validation of the web services compositions and workflows shows how to model transactional behavior and involves the verification of behavioral properties. A first approach based on the Event_B method and refinement was proposed in [4]. In this paper, we discuss the use of the Event_B model obtained by this approach in the case of transactional web services and their composition.

2 Event_B for analyzing transactional web services

Our idea is to provide assistance to BPEL developers using the Event_B method B, more precisely, providing a methodology for detecting the BPEL process parts that handle critical resources. At the begining, the developer builds its BPEL process without taking into account the transactional constraints. Thereafter, the obtained BPEL process is translated into Event_B model using the

[1] OASIS approved BPEL as a standard for Web Service composition

rules defined in [4]. This step is automatically performed by the BPEL2B tool [5]. The transactional properties and the properties related to the consistencies of resources used by the BPEL process are manually expressed in the form of invariant in the INVARIANTS clause of the obtained Event_B model.

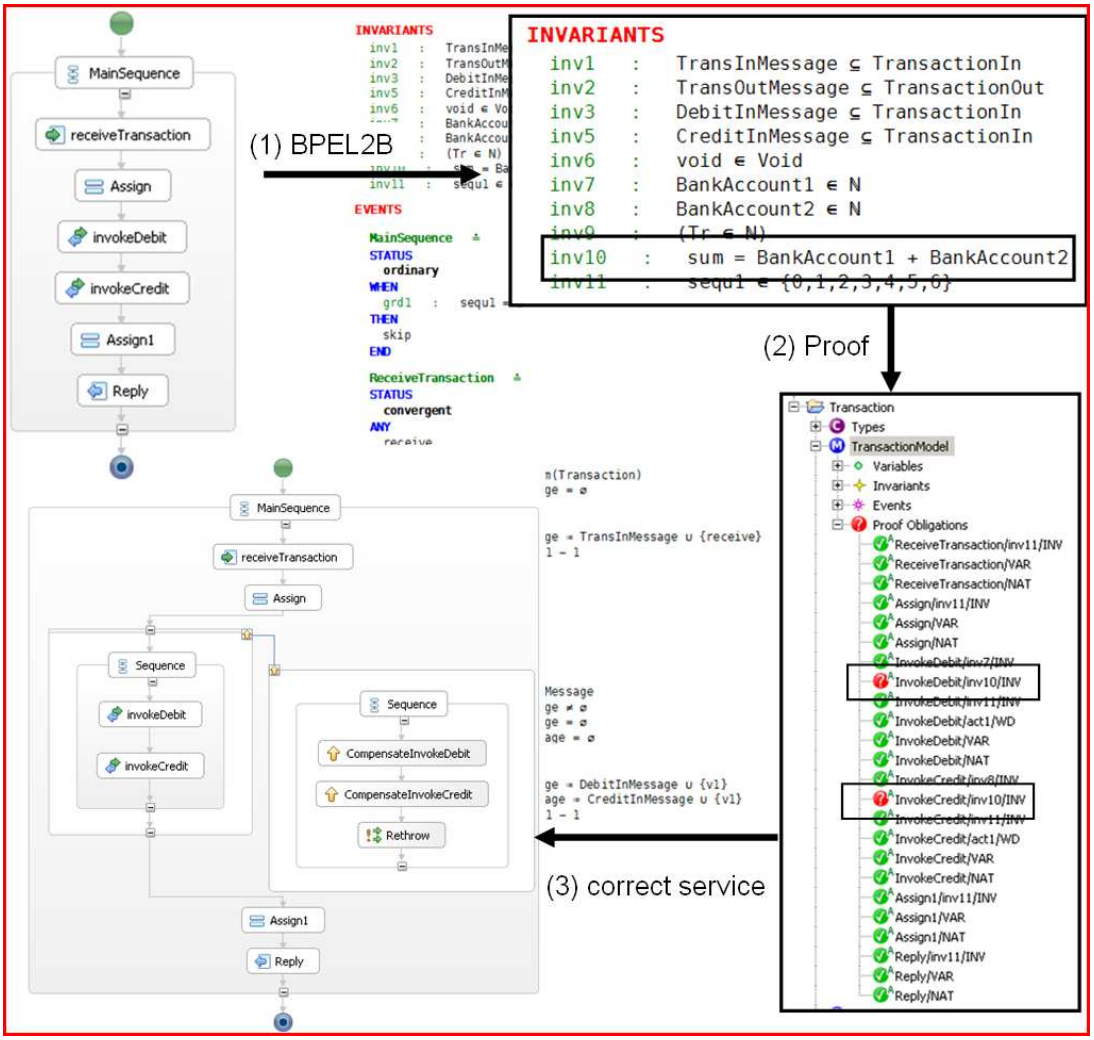


Fig. 1. The Event_B invariant to detect BPEL transactional activities.

When using the RODIN platform [6], proof obligations (PO) are automatically generated and are proved by the RODIN prover or semi-automatically by the designer. Some of these POs related to invariants involving the transactional properties not provable because triggering these events

separately violates the consistency invariants. Then, BPEL activities related to the events responsible for these POs are detected and isolated in the BPEL *scope* element (see section 12 of [1]). This BPEL element allows the designer to define a particular BPEL part on which specific mechanisms only apply to this isolated part. In our case, for transactional BPEL parts, the mechanisms for fault and compensation handling are applied to the *scope* element and the *isolated* attribute of the corresponding scope is set to TRUE. As a consequence, the execution of this part is isolated by the tools, and at the same time consistency of the resources used by these activities is guaranteed.

On figure 1, an example of a transactional process which makes a bank transfer between two bank accounts. The invariant *inv10 : sum = BankAccount1 + BankAccount2* expresses the fact that the sum of two bank accounts (*BankAccount1* and *BankAccount2*) is always constant. The POs, associated to the preservation of this invariant (squares on figure 1), generated by the events *InvokeDebit* and *InvokeCredit* for this invariant are not provable. This helps the designer to isolate the activities *InvokeDebit* and *InvokeCredit* on the BPEL process.

From a methodological point of view, our approach relies on the following steps.

1- Translate the BPEL model into Event_B applying the approach described in [4].
2- Introduce in the Event_B model the relevant invariants related to the transactional aspects.
3- Isolate the events of the Event_B model whose POs, associated to the introduced invariant of step 2, are not provable.
4- Re-design the BPEL model of step 1 by introducing a BPEL *scope* embedding the events identified at step 3 and a compensation/fault handler component.
5- Apply step 1.

This step based approach is applied until the associated Event_B model is free of unproved PO.

3 Conclusion

In this paper, we have sketched a methodology showing how the Event_B model obtained by the approach described in [4] can be used to prove web services transactional properties. Transactional services that access and manage critical resources are isolated in a *scope* elements with compensation and fault handlers. When modelling fault and compensation handlers by a set of events, it becomes possible to model and check the properties related to transactional web services. These properties are encoded in the INVARIANTS clause in order to guarantee consistency of the manipulated resources.

References

1. Jordan, D., Evdemon, J.: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Standard (April 2007) http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
2. He, Y., Zhao, L., Wu, Z., Li, F.: Formal Modeling of Transaction Behavior in WS-BPEL. In: International Conference on Computer Science and Software Engineering (CSSE 2008). (2008)
3. Guidi, C., Lucchi, R., Mazzara, M.: A Formal Framework for Web Services Coordination. In: ENTCS, Volume 180, Issue 2, Pages 55-70. (June 2007)
4. Aït-Sadoune, I., Aït-Ameur, Y.: A Proof Based Approach for Modelling and Veryfing Web Services Compositions. In: 14th IEEE International Conference on Engineering of Complex Computer Systems ICECCS'09, Potsdam, Germany (2-4 June 2009) 1–10
5. Aït-Sadoune, I., Aït-Ameur, Y.: From BPEL to Event_B. In: Integration of Model-based Formal Methods and Tools Workshop (IM_FMT 2009), Dusseldorf, Germany (16 february 2009)
6. ClearSy: Rodin (2007) http://www.methode-b.com/php/travaux_r&d_methode_b_projet_RODIN_fr.php.

Using Event-B to Verify the Kmelia Components and their Assemblies

Pascal André, Gilles Ardourel, Christian Attiogbé and Arnaud Lanoix

COLOSS Team
LINA CNRS UMR 6241 - University of Nantes
{firstname.lastname}@univ-nantes.fr

Component-based software engineering is a practical approach to address the issue of building large software by combining existing and new components. However, building reliable software systems from components requires to verify the consistency of components and the correctness of their assemblies. In this context we proposed an abstract and formal model, named Kmelia [1,2], with an associated language to specify components, their provided and required services and their assemblies; we also developed a framework named COSTO [3] and re-used some verification tools [1,4] to study the Kmelia specifications.

A Kmelia component is equipped with invariants and with pre/post-conditions defined on services. A Kmelia assembly defines a set of links between required and provided services of various components, with respect to their pre/post-conditions. Our main concern is to establish the correctness of Kmelia components and their assemblies. Among the formal analysis necessary to ensure complete correctness, we consider: (i) the component invariant consistency vs. pre-/post-conditions of services; (ii) the Kmelia assembly link contract correctness, that relates services which are linked in the assemblies. We use the notion of contract as in the classical works and results such as *design-by-contract* [5] or *specification matching* [6]: on the one hand the pre-condition of a required service is stronger than the pre-condition of the linked provided service; on the other hand the post-condition of the provided service is stronger than the post-condition of the linked required service. This motivates the choice for using Event-B and the Rodin framework to check the consistency of Kmelia components and the correctness of their assembly contracts, by discharging generated proof obligations.

Figure 1 gives an overview of the necessary Event-B models, generated from parts of the Kmelia specifications we want to verify. We design Event-B patterns to guide the translation and build the necessary proof obligations.

In order to verify the Kmelia invariant consistency rules, we systematically build appropriate Event-B models, by translating the necessary Kmelia elements in such a way that the Event-B proof obligations (POs) correspond to the specific rules we needed to check at the Kmelia level. Three kinds of Event-B models are to be extracted:

- a first Event-B model C_obs corresponds to the observable part of the Kmelia component ;
- another Event-B model (C) is built as a refinement of the previous one C_obs to consider the whole component, not only its observable part;
- for each required service, an Event-B model A_servR is built.

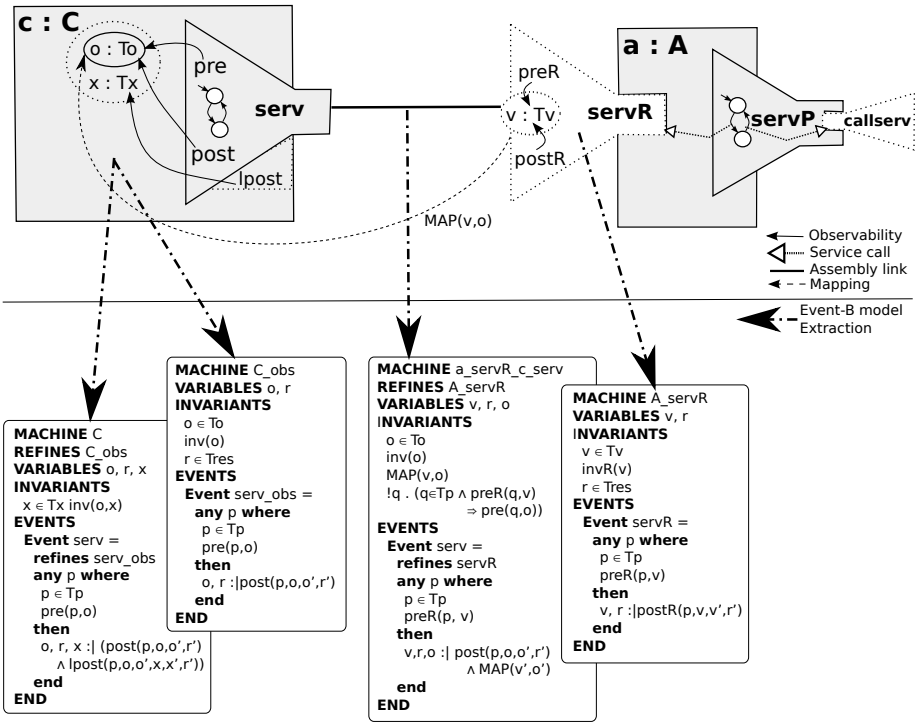


Fig. 1. Event-B Extraction patterns

We describe how the proofs of the Event-B models are linked with the attempted proofs at the Kmelia level. As an illustration, consider the generated POs about the invariant preservation [7] by the event serv_obs:

$$
\begin{array}{l}
o \in To \wedge inv(o) \ \wedge r \in Tres \ \wedge p \in Tp \\
\wedge \ pre(p,o) \quad \wedge post(p,o,o',r') \\
\Rightarrow \quad o' \in To \wedge inv(o') \quad \wedge r' \in Tres
\end{array}
$$

This corresponds exactly to the intended invariant consistency of the observable part at the Kmelia level.

For each assembly link between a required service servR and a provided one serv, we build an Event-B model as a refinement of the Event-B model previously generated for the required service servR. The observable variables of the provided service are added and the invariant is completed with the mapping MAP(v,o). Then Event-B refinement proof obligations are generated and discharged:

1. *Invariant preservation*

$$
\begin{array}{l}
v \in Tv \wedge inv(v) \quad \wedge res \ \in Tres \ \wedge \\
o \in To \wedge inv(o) \ \wedge MAP(v,o) \wedge \forall \, q \ . \ (\ q \in Tp \wedge preR(q,v) \Rightarrow pre(q,o) \ ) \\
p \in Tp \wedge preR(p,v) \ \wedge
\end{array}
$$

$$
\begin{array}{l}
\mathrm{post}(p,o,o',r') \quad \wedge \mathrm{MAP}(v',o') \\
\Rightarrow \\
o' \in \mathrm{To} \wedge \mathrm{inv}(o') \wedge \mathrm{MAP}(v',o') \wedge \forall\, q \;.\; (\; q \in \mathrm{Tp} \wedge \mathrm{preR}(q,v') \Rightarrow \mathrm{pre}(q,o') \;)
\end{array}
$$

With an $\wedge$-elimination, we consider $\forall q.\ (q \in \mathrm{Tp} \wedge \mathrm{preR}(q,v') \Rightarrow \mathrm{pre}(q,o'))$ in the right hand side. Then, the use of $p \in \mathrm{Tp} \wedge \mathrm{preR}(p,v)$ in the left hand side, combined with MAP(v',o') enables us to conclude that $\mathrm{pre}(q,o')$ holds.

2. *Action simulation*

$$
\begin{array}{l}
v \in \mathrm{Tv} \wedge \mathrm{inv}(v) \quad \wedge \mathrm{res} \in \mathrm{Tres} \;\wedge \\
o \in \mathrm{To} \wedge \mathrm{inv}(o) \wedge \mathrm{MAP}(v,o) \wedge \forall\, q \;.\; (\; q \in \mathrm{Tp} \wedge \mathrm{preR}(q,v) \Rightarrow \mathrm{pre}(q,o) \;) \\
p \in \mathrm{Tp} \wedge \mathrm{preR}(p,v) \;\wedge \\
\mathrm{post}(p,o,o',r') \quad \wedge \mathrm{MAP}(v',o') \\
\Rightarrow \\
\exists\, v'.\ \mathrm{postR}(p,v,v',r')
\end{array}
$$

These POs establish the Kmelia assembly link contract correctness rules.

The refinement technique of Event-B is used to manage both the structuring of the generated Event-B models and also the proofs to be discharged. Yet we have applied the technique to small and medium size case studies. Using classical B to validate components assembly contracts has been investigated in [8]. Our approach is quite similar with respect to the use of the refinement to check the assembly, but we start from complete component descriptions and target Event-B to prove properties. Compared with existing works, our work contributes at the level of correct-by-construction components and also at the level of the consistency of component assemblies. The results of the current work constitute one more step for rigorously building components and assemblies using the Kmelia framework.

References

1. Attiogbé, C., André, P., Ardourel, G.: Checking Component Composability. In: 5th Intl. Symposium on Software Composition, SC'06. Volume 4089 of LNCS., Springer (2006)
2. André, P., Ardourel, G., Attiogbé, C.: Defining Component Protocols with Service Composition: Illustration with the Kmelia Model. In: 6th International Symposium on Software Composition, SC'07. Volume 4829 of LNCS., Springer (2007)
3. André, P., Ardourel, G., Attiogbé, C.: A Formal Analysis Toolbox for the Kmelia Component Model. In: Proceedings of ProVeCS'07 (TOOLS Europe). Number 567 in Technichal Report, ETH Zurich (2007)
4. André, P., Ardourel, G., Attiogbé, C., Lanoix, A.: Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies. In: 6th International Workshop on Formal Aspects of Component Software(FACS 2009). LNCS (2009) to be published.
5. Meyer, B.: Applying "design by contract". IEEE COMPUTER **25** (1992) 40–51
6. Zaremski, A.M., Wing, J.M.: Specification Matching of Software Components. ACM Transaction on Software Engeniering Methodolology **6**(4) (1997) 333–369
7. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundamenta Informaticae **77**(1-2) (2007) 1–28
8. Lanoix, A., Souquières, J.: A Trustworthy Assembly of Components using the B Refinement. e-Informatica Software Engineering Journal (ISEJ) **2**(1) (2008) 9–28

Code Synthesis for Timed Automata: A Comparison using Case Study

Anaheed Ayoub[1], Ayman Wahba[2], Ashraf Salem[1], Mohamed Sheirah[2]

[1] Mentor Graphics Egypt
[2] Ain Shams University

Abstract. In this paper we apply two different code synthesis approaches to an industrial case study. Both approaches automatically generate the implementation code from the timed automata model. One of these approaches is based on the using of B-method and its available code generation tools. We compare the resultant implementation code, using these two approaches, by mean of simulation.

Keywords: Code Synthesis, Timed Automata, Production Cell

1 Introduction

Timed automata specification [1] is one of the successful approaches for modeling real time systems. There is a need to automatically transform the verified timed automata model to executable code (the implementation). By using well-defined transformation steps then the generated code will be correct-by-construction.

To the best of our knowledge, there are only two available approaches to automatically generate implementation code from timed automata model. One is described in [2]. Also this approach is implemented and attached to TIMES tool [3]. We will call this approach "TIMES approach".

The other approach is based on using B-method [4]. This approach is based on automatically generate the B-model form the timed automata model and then automatically refine it into the concrete model, this concrete model is then used as an input to the available code generation tool [5] that generates the actual program code. The details of this approach are described in [6]. We will call this approach "B-method approach".

In this paper we compare these two code synthesis approaches. We use an industrial case study (named production cell) for this comparison. The correctness of the resultant implementation code is verified by mean of simulation.

2 The Comparison

We select the model of the production cell to be used as a case study for the comparison between these two approaches. The production cell is an industrial

process that is used to forge metal plates (or blanks) in a press [7, 8]. The full description of the timed automata model of this production cell can be found in [9]. The production cell model applied to TIMES approach is exactly the same one applied to the B-method approach. We verified this model against its properties written as TCTL (Timed Computation Tree Logic) formulas as given in [9].

The B-method approach generates platform independent code [6]. So we select the generated code using TIMES to be platform independent too for the comparison purpose.

For the B-method approach, we use the deterministic semantic of timed automata which is used for TIMES code generation as given in [2]. This semantic controls the selection of the next executed function. This deterministic semantic includes,

- The run-to-completion semantic, which means that as long as there is an enabled action transition then it will be taken before the time progression. In other words the time will progress only when no more action transitions are enabled.
- The non-determinism between action transitions is resolved by defining priorities for the action transitions. So if several transitions are enabled the one with the highest priority (written first) is taken.

The using of this deterministic mechanism is generally not needed for the code generated by the B-method approach. But we use it as it is the implemented mechanism for the TIMES approach. So we select to use it for comparison purpose.

By running the code generated using the B-method approach, it works fine as far as we run and no property violation could be found. On the other hand the code generated using TIMES approach runs successfully for the first 10 action transitions and then it progresses the time infinitely. This means that the system deadlocked, so it violates the first property of the model. While the first property is to grantee that the system is deadlock free.

By trying to run the same sequence of transitions - that generates the deadlock - on the timed automata model using TIMES tool, we have found that this sequence is not a valid one. By more investigation we have found that the reason behind this deadlock is that neither the committed states [10] nor the urgent ones [10] are taken into consideration during the code generation for the TIMES approach. And so the priority that is given to the enabled action transitions violates these states settings. This mishandling drives the system into the deadlock state.

The committed and urgent states are handled in the B-method approach. And so the generated code using the B-method approach doesn't suffer from this weakness. The overall flow of this comparison is shown in Figure 1.

3 Conclusions

In this paper we compared two code synthesis approaches. Both approaches automatically generate the implementation code from the timed automata model. The first approach is implemented as a part of TIMES tool while the other approach takes advantages of the B-method features and reliable tools. We used the production cell case study for this comparison. The correctness of the resultant implementation code is verified by mean of simulation.

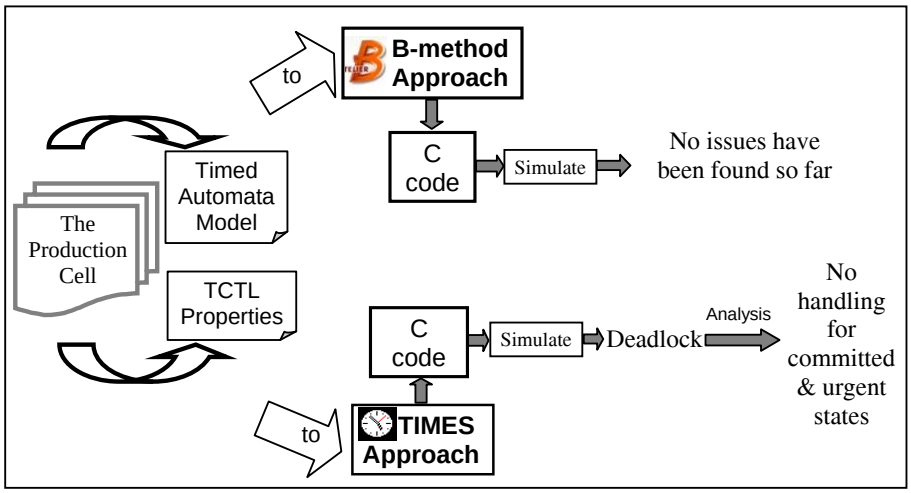


Figure 1: The Overall Flow

The comparison gave a result that the approach based on the using of B-method generates a verified code (by mean of simulation) and handles more timed automata features.

References

1. Rajeev Alur and David L. Dill., Automata for modeling real-time systems. In Proceedings, Seventeenth International Colloquium on Automata, Languages and Programming. England, pp 322–335. (1990)
2. Tobias Amnell, Elena Fersman, Paul Pettersson, Hongyan Sun, Wang Yi. Code Synthesis for Timed Automata. In Nordic Journal of Computing (NJC), volume 9, number 4. (2002)
3. http://www.it.uu.se/research/group/darts/times/papers/manual.pdf
4. J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press. (1996)
5. http://www.tools.clearsy.com/index.php5?title=Tutorial_ComenC
6. Anaheed Ayoub, Ayman Wahba, Ashraf Salem, Mohamed Taher, Mohamed Sheirah. “Automatic Code Generation from Verified Timed Automata Model”. To be appear in proceeding of IADIS Applied Computing, Rome, Italy. (2009)
7. K. Feyerabend and R. Schlor. "Hardware synthesis from requirement specifications". In Proceedings EURO-DAC with EURO-VHDL 96. IEEE Computer Society Press. (1996)
8. D. R. W. Holton. "A PEPA Specification of an Industrial Production Cell". The Computer Journal, vol. 38, No. 7, pages 542-551. (1995)
9. Anaheed Ayoub, Ayman Wahba, Ashraf Salem, and Mohamed Sheirah. "TCTL-Based Verification of Industrial Processes". In proceeding of FORUM on Specification &and Design Languages, FDL′03. Pages 456-468. Frankfurt, Germany. (2003)
10. http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf

Architecture as an Independent Variable for Aspect-Oriented Application Descriptions

Hamid Bagheri and Kevin Sullivan

University of Virginia
151 Engineer's Way
Charlottesville, VA 22903 USA
{hb2j,sullivan}@virginia.edu

Abstract. In previous work[1], we demonstrated the feasibility of formally treating architectural style as an independent variable. Given an application description and architectural style description in Alloy [3], we *map* them to software architecture description that refines the given application in conformance with the given style. This paper extends our earlier work to aspect-oriented structures. We describe an aspect-enabled application description style, a map taking application descriptions in this style to pipe-and-filter architectures, and *A2A*, a tool that converts Alloy-computed architectures to the AspectualACME architectural descriptions language[2].

1 Application Description

We present our idea in an example: a mapping of Parnas's KWIC [4], enhanced with a logging concern, to an architectural description in the *pipe-and-filter* style [4, 5]. We describe KWIC in a *composition of functions application style*. We now define a new application description abstraction called *AspectualFunction* that extends the traditional function with crosscutting relations to other functions. *Before*, *around* and *after* annotations specify these relations. *Before* states that an AspectualFunction is executed before an affected element. *Around* allows to skip execution of the affected function. *After* executes after an affected function terminates. The left figure outlines our aspect-oriented KWIC description in Alloy. It has four functions (input, cs, alph, output), and an aspectual function, *logging*, with crosscutting relations to the input and output functions.

2 Architectural Map

We now outline an Alloy implementation of an architectural map taking Alloy-encoded application descriptions in this aspect-enabled style to architecture descriptions in the pipe-and-filter style. To represent a map, we extend a traditional architectural style description (in Alloy) with predicates for mapping application descriptions in a given style to architectural descriptions in the given style. These predicates take application descriptions as parameters (such as the KWIC

```
...
one sig input extends Function{}
one sig cs extends Function{}
one sig alph extends Function{}
one sig output extends Function{}
one sig KWIC extends SystemFunctions{}{
   functions.first =  input
   functions.rest.first = cs
   functions.rest.rest.first = alph
   functions.last =  output
}
one sig logging extends AspectualFunction{}{
   before = output &&
   after = input &&
   around = none
}
...
```

```
handleAspectualFunctions[funs: set Function] {
...
all af:AspectualFunction | (af in funs) => {
        one acomp:Acomponent, p:Port,
        acon:Aconnector, cRole:CrosscuttingRole |
        acomp.handle = af  &&
        p in acomp.ports &&
        cRole in acon.crosscuttingRoles &&
        cRole->p in acomp.~components.attachments &&
        all f:Function | (f in af.after) => {
                one c:Component, outPort:Output , bRole:BaseRole |
                bRole in acon.baseRoles &&
                f.~handle = c &&
                outPort in c.ports &&
                cRole->bRole in acon.glue.after &&
                bRole->outPort in acomp.~components.attachments  }
... }
```

Fig. 1. Left: enhanced KWIC description (elided) in Alloy, Right: part of the map predicate represented in Alloy.

structure illustrated above), and define relationships required to hold between them and computed architectural descriptions. Given an application description, and a map, Alloy computes corresponding architectural descriptions guaranteed to *conform* to the given architectural style.

Figure 1 at right presents the *handleAspectualFunctions* mapping predicate. This parameterized predicate along with other predicates (elided for space) accepts application descriptions in the aspect-enabled composition-of-functions style and produces pipe-and-filter architecture description. It specifies that for each aspectual function there is both a component in the architectural description that handles it and an aspectual connector. The crosscutting role of the aspectual connector is connected to the component's port. For each affected function, after which the aspectual function should be executed, there is a *BaseRole* attached to the output port of the component that handles the affected function. There is also analogous code for before and around situations.

3 Architecture Description

We use the Alloy Analyzer to compute architecture descriptions, represented as satisfying solutions to the constraints of a map given an application description. The *A2A* transformer application then converts the computed output to an architecture description in a traditional architecture description language (ADL): here, AspectualACME. Figure 2 illustrates the AspectualACME description of our KWIC with logging. *DataSource*, a specific type of Filter, handles the *input* function. Its output port is connected to *Pipe0*. *Filter1* handles the *CS* function. Its input and output ports are connected to *Pipe0* and *Pipe1*, respectively. Similarly, the other filters handle *alph* and *output* functions. *Logging* affects *DataSource* and *DataSink*. The compositions of the AComponent0 component that handles logging, with the DataSource0 and DataSink0 components are modeled by the AConnector0 aspectual connector. It connects the output port of DataSource0 as well as the input port of DataSink0 with the acPort0.

```
...
Component AComponent0 = {
        Port acPort0;
        Property handle = "logging";
}
Connector AConnector0 = {
        Base Role aBaseRole0, aBaseRole1;
        Crosscutting Role aCrosscuttingRole0;
        Glue { aCrosscuttingRole0 after aBaseRole0;
               aCrosscuttingRole0 before aBaseRole1;
        }
}
Attachment DataSource0.output to Pipe0.source;
Attachment Filter1.input to Pipe0.sink;
Attachment Filter1.output to Pipe1.source;
Attachment Filter0.input to Pipe1.sink;
Attachment Filter0.output to Pipe2.source;
Attachment DataSink0.input to Pipe2.sink;
Attachment AComponent0.acPort0 to AConnector0.aCrosscuttingRole0;
Attachment DataSource0.output to AConnector0.aBaseRole0;
Attachment DataSink0.input to AConnector0.aBaseRole1;
...
```

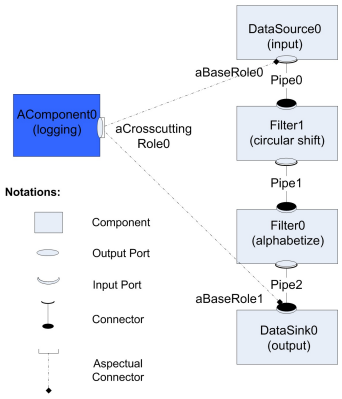


Fig. 2. KWIC AspectualACME Description with logging.

The glue clause of AConnector0 specifies that the element bound to the crosscutting role of aCrosscuttingRole0 acts after the execution of the element bound to the aBaseRole0, and acts before the execution of the element bound to the aBaseRole1.

4 Discussion

Software architecture researchers have long assumed that architecture-independent application descriptions can be mapped to architectures in many styles, that results vary in quality attributes, and that the choice of a style is driven by consideration of such attributes. The contribution of our earlier work was to show that we can make this idea precise and computable. This paper extends that work to the case of crosscutting concerns.

References

1. H. Bagheri, Y. Song, and K. Sullivan. Architecture as an independent variable. Technical report CS-2009-11, University of Virginia Department of Computer Science, Nov. 2009.
2. A. Garcia, C. Chavez, T. Batista, C. Santanna, U. Kulesza, A. Rashid, and C. Lucena. On the modular representation of architectural aspects. In *Proceedings of the European Workshop on Software Architecture*, pages 82—97, Nantes, France, 2006. Lecture Notes in Computer Science.
3. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256—290, 2002.
4. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053—1058, 1972.
5. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

Integrating SMT-solvers in Z and B Tools

A. C. Gurgel[*1], V. G. Medeiros Jr.[2], M. V. M. Oliveira[1] and D. B. P. Dharbe[1]

[1] Departamento de Informtica e Matemtica Aplicada, UFRN, Brazil
[2] Instituto Federal de Educaão, Cincia e Tecnologia, IFRN, Brazil

An important frequent task in both Z [14] and B [1] is the proof of verification conditions (VCs). In Z and B, VCs can be predicates to be discharged as a result of refinement steps, some safety properties (preconditions) or domain checking. Ideally, a tool that supports any Z and B technique should among other tasks, automatically discharge as many VCs as possible. Here, we present ZB2SMT [3], a `Java` package designed to clearly and directly integrate both Z and B tools to the satisfiability module theory (SMT) solvers such as veriT [3], a first-order logic (FOL) theorem prover that accepts the SMT syntax [12] as input. By having the SMT syntax as target we are able to easily integrate with further eleven automatic theorem provers that are also compatible like [5,2,7].

veriT provides an open framework to generate certifiable proofs, having a decent efficiency [3] that does not compromise the performance of the tools in usual developments. Its input format is the SMT-LIB language extended with macro definitions. This syntactic facility uses lambda notation and is particularly useful to write formulas containing simple set constructions. This feature enhances the ability of veriT to handle sets, making the solver an interesting tool in formal development efforts in set-based modelling languages.

This prover is used by Batcave [9], an open source tool that generates VCs for the B method. Batcave has a friendly graphic interface and supports B specification with representation in XML format. It uses a parser from the JBTools [13] that is composed by the B Object Library (BOL).

CRefine [11] is a tool that supports the use of the *Circus* refinement calculus. Circus [4] is a concurrent language tailored for refinement that combines Z with CSP [6] and the refinement calculus[10]. CRefine allows the automatic application of refinement laws and discharge of VCs. Much of the VCs generated to validate the refinement law applications, are based on FOL predicates. Hence, CRefine uses veriT to automatically prove such predicates.

In order to allow reuse, we have developed the package ZB2SMT, which integrates elements of Z and B predicates in a common language and transforms these predicates into SMT syntax. ZB2SMT uses an extension of BOL to represent B predicates. On the other hand, the package uses a framework provided by the Community Z Tools (CZT) [8], an ongoing effort that implements tools for standard Z, to represent Z predicates.

In ZB2SMT, Z predicates are converted to B predicates, using the extension of BOL. The extension is needed due to the fact that there are Z operators that do not exist in BOL like the symmetric difference operator. Extending BOL by

[*] The ANP supports the work of the author through the prh22 project.
[3] Freely available at `http://www.consiste.dimap.ufrn.br/projetos/zb2smt`.

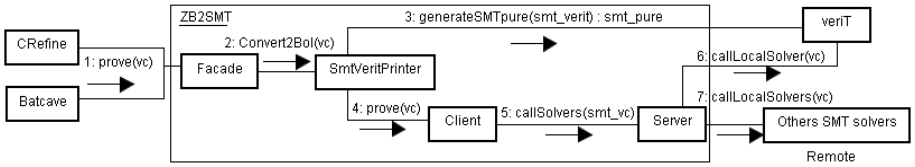


Fig. 1. Collaboration Diagram of ZB2SMT elements

including these missing operators, we improve the set of predicates that can be treated by ZB2SMT. These predicates are translated into a SMT syntax and written to a file that contains the predicate and some elements such as types of variables, operators definition and set properties that are described over the macro feature. The SMT file is sent to veriT which yields a boolean value for the predicate. On successful evaluation of the predicate, the resulting value is returned. If, however, the evaluation is not successful, veriT can be used to return a SMT file that may be sent to others SMT solvers. This kind of file is a bit different from the original input of veriT since others SMT solvers do not have the peculiarity of macros definition. ZB2SMT allows an integration with the others SMT provers using the conversion from SMT-verit to SMT-pure by veriT.

The application of formal development to large programs generally produces a great amount of VCs. Perform an automatic proof module in only one processor may be impracticable. In order to improve the performance of the proof system, the ZB2SMT has a module that can call different instances of theorem provers on different computers, using socket and `Java`'s thread. The flow of execution of the module in ZB2SMT is illustrated in Figure 1. For conciseness, Figure 1 does not show the conversion from Z to B predicates.

This module has two parts: the client with the information about each possible instance of theorem prover, which can be local or remote, and the server with the theorem prover installed locally. The user creates a configuration for each instance of theorem prover in a file. It contains the following information: path of theorem prover, parameters and the host machine.

The parallelization process replicates VCs and tries to solve by different strategies. Each instance of theorem prover has its own set of specifics strategies and parameters to try to solve the VCs. Thus, the user can create and explore different strategies possibly making the proof process more efficient.

The motivation for our work is to provide a direct verification engine to discharge VCs from Z, B or extensions of their tools. ZB2SMT has been effective and promising in the first experiences in CRefine and Batcave. It can be directly used, like a black box, by tools that work with the CZT framework for Z or B tools which use the BOL library. Furthermore, ZB2SMT offers an easy way to get SMT files from B or Z predicates. Currently, we are embedding, by adjusting parameters and path configurations, others SMT solvers in ZB2SMT.

The parallelism inside ZB2SMT has been an important feature. It improves the proof process by allowing different strategies to be performed in parallel, reducing the verification time. However, the performance of our system can be improved even more by incorporating a predicate classifier. It would classify the predicate and select the best available SMT solver to prove it, since some SMT solvers are more efficient in certain types of predicates.

Acknowledgments. This work was partially supported by INES (www.ines.org.br), funded by CNPq grant 573964/2008-4 and by CNPq grants 553597/2008-6, 550946/2007-1, and 620132/2008-6.

References

1. J. R. Abrial. *The B Book: Assigning Programs to Meanings.*, volume 1 of *1*. Cambridge University Press, United States of America, 1 edition, 1996.
2. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.
3. Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In *CADE-22 (Int'l Conf. Automated Deduction*, pages 151–156, 2009.
4. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A refinement strategy for *Circus*. *Formal Aspects of Computing*, **15**(2–3):146–181, 2003.
5. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. pages 337–340. 2008.
6. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
7. Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. 2009.
8. P. Malik and M. Utting. CZT: A Framework for Z Tools. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB*, volume **3455** of *Lecture Notes in Computer Science*, pages 65–84. Springer, 2005.
9. E. S. Marinho, V. G. Medeiros Jr, Cláudia Tavares, and David Déharbe. Um ambiente de verificação automática para o método B. In A. C. V. Melo and A. Moreira, editors, *SBMF 2007: Brazilian Symposium on Formal Methods*, 2007.
10. C. Morgan. *Programming from Specifications.* Prentice-Hall, 1994.
11. M. V. M. Oliveira, A. C. Gurgel, and C. G. de Castro. CRefine: Support for the *Circus* Refinement Calculus. In Antonio Cerone and Stefan Gruner, editors, *6th IEEE International Conferences on SEFM*, pages 281–290. IEEE Computer Society Press, 2008. IEEE Computer Society Press.
12. Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2, 2006. Available at `www.SMT-LIB.org`.
13. J. C. Voisinet. Jbtools: an experimental platform for the formal b method. In *PPPJ '02/IRE '02: Proceedings of the inaugural conference on the Principles and Practice of programming*, pages 137–139, Maynooth, County Kildare, Ireland, Ireland, 2002. National University of Ireland.
14. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.

Secrecy UML Method for Model Transformations*

Waël Hassan[1], Nadera Slimani[2], Kamel Adi[2], and Luigi Logrippo[2]

[1] University of Ottawa, 4051D-800 King Edward, Ottawa, Ontario, K1N-6N5
[2] Universit du Qubec en Outaouais, 101 Rue St-Jean-Bosco, Gatineau, Qubec, J8X 3X7
wael@acm.org, {slin02, Kamel.Adi, luigi}@uqo.ca

Abstract. This paper introduces the subject of secrecy models development by transformation, with formal validation. In an enterprise, constructing a secrecy model is a participatory exercise involving policy makers and implementers. Policy makers iteratively provide business governance requirements, while policy implementers formulate rules of access in computer-executable terms. The process is error prone and may lead to undesirable situations thus threatening the security of the enterprise. At each iteration, a security officer (SO) needs to guarantee business continuity by ensuring property preservation; as well, the SO needs to check for potential threats due to policy changes. This paper proposes a method that is meant to address both aspects. The goal is to allow not only the formal analysis of the results of transformations, but also the formal proof that transformations are property preserving. UML is used for expressing and transforming models [1], and the Alloy analyzer is used to perform integrity checks [6].

Keywords: Model transformation, Property preservation, SUM, Alloy, UML.

1 Introduction

Governance requirements dictate a security policy that regulates access to information. This policy is implemented by means of secrecy models[3] that establish the mandatory secrecy rules for the enterprise. For example, a secrecy rule may state: *higher-ranking officers have read rights to information at lower ranks.* In addition, *Business policies* may specify instances such as: *user A has access to department M.* Hence, an enterprise governance system is composed of a combination of *secrecy model rules* and *business policies.*

Automation helps reduce design errors of combined and complex secrecy models [3]. However, current industry practices do not include precise methods for constructing and validating enterprise governance models. Our research

* This work has been funded in part from grants of the Natural Sciences and Engineering Research Council of Canada and CA Labs.

[3] We concentrate in this paper on secrecy, which is one of several aspects of security.

proposes a formal transformation method to construct secrecy models by way of applying transformations to a base UML model. It provides the advantage of formal validation of constructed models. For example, starting from an initial model called Base Model (BM), with only three primitives: Subject/Verb/Object, we can generate –by transformation functions– RBAC0 (Role Based Access Control model) in addition to a SecureUML model.

By way of examples we intend to show that our method is potentially useful for building different types of secrecy models. By means of formal analysis we intend to show that a SO will be able to validate a resultant model for consistency in addition to detecting scenarios resulting from unpreserved properties.

In this paper, we present our method in section 2. In section 3, we show examples that illustrate our approach with application results. Finally, we conclude this paper, in section 4, and discuss the future work and perspectives.

2 Secrecy UML Method (SUM)

SUM serves as a systematic method to construct secrecy properties for enterprise governance. Starting from a generic UML model, that we call *base model* (BM) and a set of transforming operations (TOs) (see Fig. 1). The operations are conjectured to be *property-enriching* as well as *property-preserving* and are applied to achieve a *resultant model* (RM). In Fig. 1, several rectangles labelled: Specialize, Aggregate, Compose, Split Right, Split Left, Reflex, Tree Macro, and Graph Macro. These rectangles represent the transformation operations. Each of the transformation operations takes as input a class labelled 'input' –shown using a grey shading– and produces the respective output –as shown. The TOs modify the base model iteratively in a way that, in practice, the resultant model is customized and used to govern security or privacy properties of a particular enterprise.

We use a first order logic formalism to: represent the base model, the transformation operations, and the resultant model. We show that it is possible to validate the resultant model for consistency using logic analysers. The translation from the UML language to a logic analyser language can be done through the use of a specialised secrecy modelling language called SML [4]. SML provides a component view of Alloy code that is conjectured to facilitate the representation of secrecy models. Alternatively it can be done directly through a UML to Alloy translator [6]. In all cases a transformation tool can be programmed to map a model to another. Alloy will validate the models for properties of model preservation and consistency.

2.1 Base-model (BM).

The base model proposed in this paper includes three primitives components: S, V and O. A subject S is a subject in the enterprise. A verb V denotes the fact that an action or right is given or denied to the subject. An object O is the data item or object to which the action or right refers.

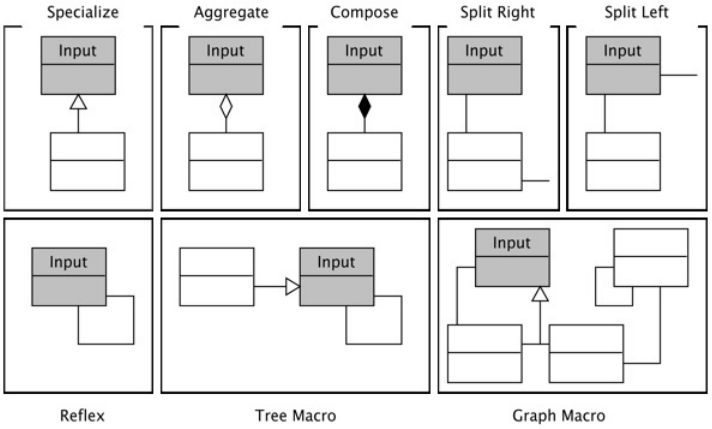


Fig. 1. Transformation operations.

2.2 Transformation Operation (TO).

A TO is an operation consuming an input and producing an output model. Our method defines the following TOs: Specialisation, Aggregation and Composition, Reflex, Split, Tree Macro, Graph Macro (See Fig. 1).

Specialisation: Following the UML definition [5] this operation extends a general class into a specific one with detailed features.

Aggregation and Composition: Aggregation and Composition describe the construction of a parent class from sub-classes, that are mandatory (sub-classes) in the case of Composition. Example of Composition: *An audit department is composed of financial and privacy audit sub-departments.* Both departments (privacy and financial audits) are necessary for the audit department to exist. On the other hand, *the set of employees consists of full-timers, part-timers, and consultants* is considered an Aggregation.

Reflex operation: A reflex transformation adds a relation to the input class. This operation is frequently used, mostly to represent a structural relation. e.g. *a node is a sibling of another node.*

Split operation: A split is often used to transform a component into a relation between two components. For example, an object can be split *into a session controlling an object.* A split can preserve all or some of the original relations of the input component. In Fig. 1, we show two kinds of split left, right, which we will detail in future work.

Tree Macro: The tree macro is useful for the construction of several secrecy models. For instance, *it can be used to represent a relation between a subject and its department or Group.*

Graph Macro: A graph macro takes an input class and creates a graph of classes. For example, *it can be used for building business processes.*

3 Examples

3.1 Transforming BM to RBAC

In this first example of transformation, we apply a set of operations so that the resultant model is similar to RBAC (Role Based Access Control model) in [2]. Fig. 2 shows the syntax representation of the RM components at each iteration.

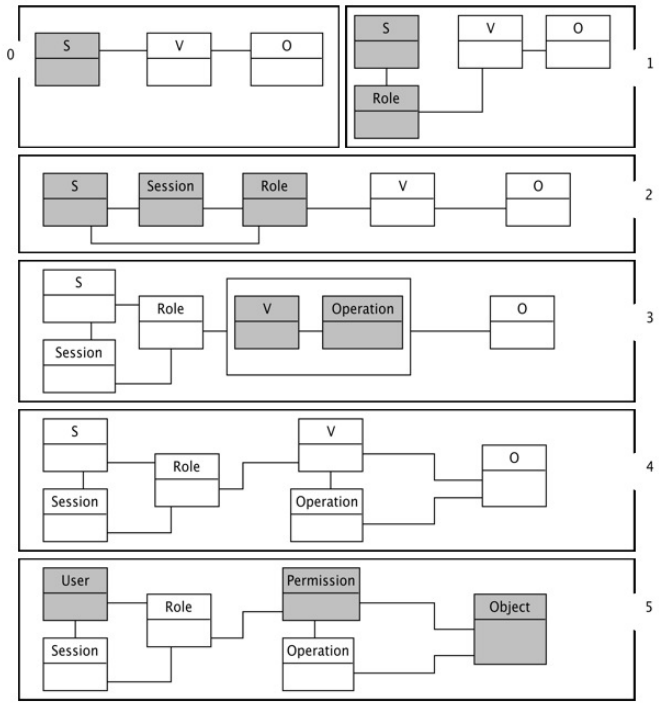


Fig. 2. Transformation steps from BM to RBAC model.

In this case, we simply apply three Split operations on both primitive components: Subject and Verb, followed by the Renaming operation, e.g. Verb becomes

Operation. Here is a list of the used successive operations, so that in the left side (the function result) we have the set of components forming the new model:

- Split(S)={S, Role}
- Split(S)={S, Session}
- Split(V)={Verb, Operation}
- Rename(S, V)={User, Permission}

3.2 Transforming BM to SecureUML

SecureUML is a security modeling language based on RBAC with refinement [3]. Fig. 3, shows the transformations needed to develop the meta-model of SecureUML.

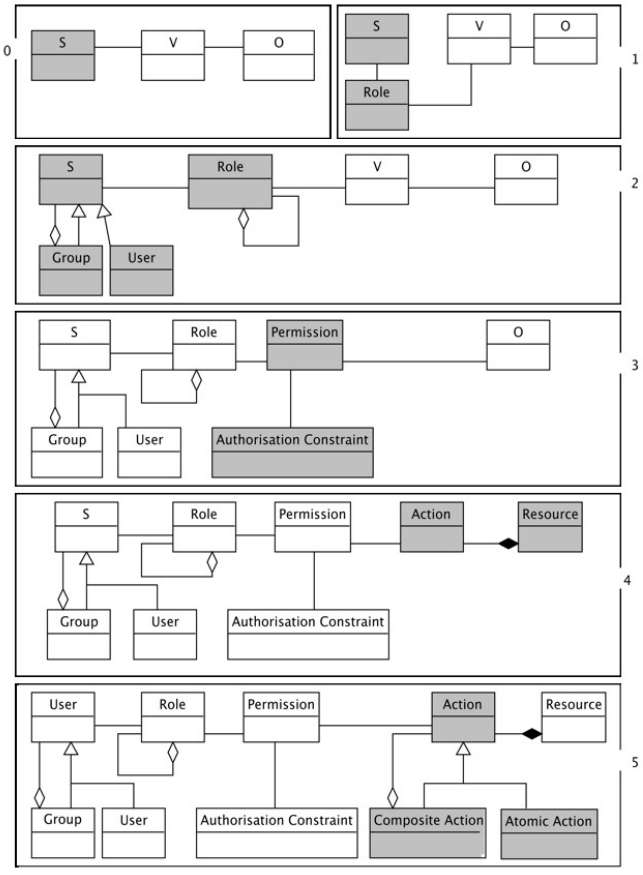


Fig. 3. Transformation steps from BM to SecureUML model.

The SecureUML is defined as an extension of RBAC. It supports: the policy constraint (using the Authorisation Constraint component), the Action hierarchy, the Specialisation of the Action in Atomic Action and Composite Action, etc. Since, SecureUML is more complicated. It requires using more than two types of TO, in the following operation list:

- Split(S)={S,Role}
- Specialise(S)={Group,User}
- Aggregate(Group)={S}
- Aggregate(Role)={Role}
- Split(V)={V, AuthorisationConstraint}
- Rename(V)={Permission}
- Compose(Resource)={Action}
- Specialise(Action)={AtomicAction, CompositeAction}
- Aggregate(AtomicAction)={Action}

4 Conclusion

In conclusion, the Secrecy UML method (SUM) supports the construction of complex secrecy models from a base-model by a disciplined transformation method. We believe that its application would be to assist a security officer in achieving the required enterprise security policy by model transformation. We will show in future publications that our technique, combining the use of UML and relational logic, allows verification of the final result using the Alloy analyser. Future work will strengthen this conjecture by proving the property-preserving characteristics of the transformations. There are several avenues for future work in this domain. We plan (i) to provide a detailed formal description of the transformation operations on a case study; (ii) to extend this paper to include the SML statements corresponding to the output model in each case; (iii) to show the ability to detect inconsistencies in the design. Finally, we foresee to create a graphical user interface module that allows a designer to transform a UML model, using SUM operations, and to create an automatic SUM to SML translator.

References

1. Berardi, D., Calvanese, D.,De Giacomo, G.: Reasoning on UML class diagrams. Artif. Intell. Essex, UK , 2005. 70-118, Elsevier Science Publ. Ltd.
2. Ray, I., Li, N., France, R., Kim, D.: Using UML to visualize role-based access control constraints. SACMAT'04. NY, USA, 2004. 115-124, ACM.
3. Basin, D., Doser, J. and Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. Softw. Eng. Methodol. NY, USA, 2006. 39-91, ACM Press.
4. Hassan, W.: Secrecy Modelling Language. http://code.google.com/p/sml-silver/. Accessed Aug 2009.
5. Evans, A., France, R. B., Lano, K. and Rumpe, B.: The UML as a Formal modelling Notation. UML'98: London, UK, 336-348, 1999. Springer-Verlag.
6. Anastasakis, K., Bordbar, B., Georg, G., Ray I.: On Challenges of Model Transformation from UML to Alloy. MoDELS 2007. 436-450.

B Model Abstraction Combining Syntactic and Semantic Methods

J. Julliand[1], N. Stouls[2], P.-C. Bué[1], and P.-A. Masson[1]

[1] LIFC, Université de Franche-Comté
16, route de Gray F-25030 Besanon Cedex
{bue, julliand, masson}@lifc.univ-fcomte.fr

[2] Université de Lyon, INRIA
INSA-Lyon, CITI, F-69621, Villeurbanne, France
nicolas.stouls@insa-lyon.fr

Abstract. In a model-based testing approach as well as for the verification of properties by model-checking, B models provide an interesting solution. But for industrial applications, the size of their state space often makes them hard to handle. To reduce the amount of states, an abstraction function can be used, often combining state variable elimination and domain abstractions of the remaining variables. This paper illustrates a computer aided abstraction process that combines syntactic and semantic abstraction functions. The first function syntactically transforms a B event system M into an abstract one A, and the second one transforms a B event system into a Symbolic Labelled Transition System (SLTS). The syntactic transformation suppresses some variables in M. This function is correct in the sense that A is refined by M. A process that combines the syntactic and semantic abstractions has been experimented. It significantly reduces the time cost of semantic abstraction computation. This abstraction process allows for verifying safety properties by model-checking or for generating abstract tests. These tests are generated by a coverage criteria such as all states or all transitions of an SLTS.

Keywords: Model Abstraction, Syntactic Abstraction, Refinement.

The full version of this short paper is available as a research report: [JSBM09].

1 Introduction

B models are well suited for producing tests of an implementation by means of a *model-based testing* approach [BJK+05,UL06] and to verify dynamic properties by model-checking [LB08]. But model-checking as well as test generation requires the models to be finite, and of tractable size. This usually is not the case with industrial applications, and the search for executions instantiated from the model frequently comes up against combinatorial explosion problems. Abstraction techniques allow for projecting the (possibly infinite or very large) state space of a system onto a small finite set of symbolic states. Abstract models

make test generation or model-checking possible in practice. We have proposed and experimented in [BBJM09] an approach of test generation from abstract models, that computes in finite time a Symbolic Labelled Transition System (SLTS) of all the behaviors of a model (with possibly an infinite concrete state space). However, it appeared that the computation time of the abstraction could be very expensive. We had replaced a problem of search time in a state graph with a problem of proof time. Indeed, computing an abstraction is performed by proving enabledness and reachability conditions on symbolic states [BPS05].

This short paper illustrates on an example our contribution [JSBM09] to reduce this proof time problem, by means of a proof free syntactic abstraction function. It works by suppressing some state variables of a model. When there are domain abstractions on the remaining state variables, a semantic abstraction that requires proof obligation checking is also performed. But it applies to a model that has been syntactically simplified.

2 Electrical System Example

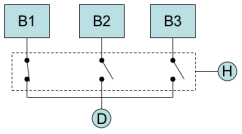


Fig. 1. Electrical System

A device D is powered by one of three batteries B_1, B_2, B_3 as shown in Fig. 1. A switch connects a battery B_i to the device D. A clock H periodically sends a signal that causes a commutation of the switches, i.e. a change of the battery that powers D. The system satisfies the three following requirements:

- Req_1: only one switch is closed at a time (i.e. there is no short-circuit),
- Req_2: there is always one switch closed, connected to a working battery,
- Req_3: a signal from the clock always changes the switch that is closed.

If a failure occurs to the battery that is powering D, the system triggers an exceptional commutation to satisfy Req_2. We assume that there are never more than two batteries down at the same time. When two batteries are down, Req_3 is relaxed and the clock signal leaves unchanged the switch that is closed.

This system is modeled by means of three variables H, Sw and Bat. $H \in \{tic,\ tac\}$ models the clock: tic means asking for a commutation and tac that the commutation has occurred. Sw models the switches: $Sw = i$ indicates that the switch i is closed while the others are opened. This modelling makes that requirements Req_1 and Req_2 necessarily hold. $Bat \in 1..3 \rightarrow \{ok, ko\}$ models the batteries, with ko meaning that a battery is down. The invariant I expresses the

assumption that at least one battery is not down by stating that $Bat(Sw) = ok$:
$I \mathrel{\widehat{=}} H \in \{tic, tac\} \wedge Sw \in 1..3 \wedge (Bat \in 1..3 \rightarrow \{ok, ko\}) \wedge Bat(Sw) = ok.$

The initial state is defined by *Init* in Fig. 2. The behavior of the system is described by four events, modeled in Fig. 2 with the primitive forms of substitutions: `Tic` sends a commutation command, `Com` performs a commutation, `Fail` simulates the failure of a battery, and `Rep` simulates the replacement of a battery.

$$
\begin{array}{l}
Init \mathrel{\widehat{=}} H,\ Bat,\ Sw\ :=\ tac,\ \{1 \mapsto ok,\ 2 \mapsto ok,\ 3 \mapsto ok\}, 1 \\
Tic \mathrel{\widehat{=}} H = tac \Rightarrow H := tic \\
Com \mathrel{\widehat{=}} \mathrm{card}(Bat \rhd \{ok\}) > 1 \wedge H = tic \Rightarrow \\
\qquad @ns.(ns \in 1..3 \wedge Bat(ns) = ok \wedge ns \neq Sw \Rightarrow H, Sw := tac, ns) \\
Fail \mathrel{\widehat{=}} \mathrm{card}(Bat \rhd \{ok\}) > 1 \Rightarrow \\
\qquad @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ok\}) \Rightarrow \\
\qquad\quad nb = Sw \Rightarrow \\
\qquad\qquad @ns.(ns \in 1..3 \wedge ns \neq Sw \wedge Bat(ns) = ok \Rightarrow \\
\qquad\qquad\quad Sw, Bat := ns, Bat <+ \{nb \mapsto ko\}) \\
\qquad\quad []nb \neq Sw \Rightarrow Bat := Bat <+ \{nb \mapsto ko\})) \\
Rep \mathrel{\widehat{=}} @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ko\}) \Rightarrow Bat := Bat <+ \{nb \mapsto ok\})
\end{array}
$$

Fig. 2. B Specification of the Electrical System

3 Syntactic Abstraction

We consider abstractions obtained by observing only a subset of variables, defined as being relevant variables. This set is built as a fixpoint, starting with chosen variables from the property to test, and growing by addition of the variables required for computing the values assigned to the relevant variables.

For example, to test the electrical system in the particular cases where two batteries are down, observing the variable *Bat* is sufficient. In [JSBM09] we define a set of transformation rules that produce a simplified model A. We prove that A is, by construction, refined by the source model M, so that it is sufficient to verify safety properties on A for them to hold on M. It is also easier to compute test cases from A than from M.

The electrical system is transformed as shown in Fig. 3 for the set of observed variables $\{Bat\}$. It is a correct B event system. The initialization only assigns the observed variable. Its value is the same as in the source model. The event `Tic` is abstracted by `skip` because its guard and its action do not refer to the observed variable. The guard of the events `Com` and `Fail`, that are in the shape of $p(\mathtt{Bat}) \wedge p'(\mathtt{H})$, are transformed in the shape of $p(\mathtt{Bat})$ because the approximation of a proposition $p(x)$ is `true`, when x is a set of non observed variables. The bound variables are considered as observed variables. The action of an event (such as `Com` for example) becomes `skip` if it only assigns non observed variables. For the `Fail` event, we only keep the assignment of the variable `Bat`. Finally, the event `Rep` is unchanged because its guard and its action only assigns the variable `Bat` and depends on the value of the bound variable `nb`.

$$
\begin{array}{ll}
Init & \widehat{=}\ Bat := \{1 \mapsto ok,\ 2 \mapsto ok,\ 3 \mapsto ok\} \\
Tic & \widehat{=}\ skip \\
Com & \widehat{=}\ \mathrm{card}(Bat \rhd \{ok\}) > 1 \Rightarrow @ns.(ns \in 1..3 \wedge Bat(ns) = ok \Rightarrow skip) \\
Fail & \widehat{=}\ \mathrm{card}(Bat \rhd \{ok\}) > 1 \Rightarrow \\
 & \quad @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ok\}) \Rightarrow Bat := Bat <+ \{nb \mapsto ko\}) \\
Rep & \widehat{=}\ @nb.(nb \in 1..3 \wedge nb \in \mathrm{dom}(Bat \rhd \{ko\}) \Rightarrow Bat := Bat <+ \{nb \mapsto ok\})
\end{array}
$$

Fig. 3. B Syntactically Abstracted Specification of the Electrical System

4 Abstraction Process

In [BBJM09] we have introduced a test generation method based on a semantic abstraction of a B model (see Fig. 4/Process A). The abstraction is computed as an SLTS according to a test purpose. The idea is to observe the state variables that are modified by the operations activated by the test purpose. The domain of the observed variables can be abstracted into a few subdomains. For example, a natural integer n can be abstracted into subdomains $n = 0$ and $n > 0$.

The two main drawbacks of this process are its time cost and the proportion of proof obligations (POs) not automatically proved. Indeed, the semantic abstraction is based on a theorem proving process [BC00]. Each unproved PO adds a transition to the SLTS that is possibly unfeasible. Hence we propose to use a syntactic abstraction in addition to the semantic one. In Fig. 4/Process B, we describe a complete abstraction process in which we combine a syntactic abstraction that eliminates some variables (see Sec. 3), with a semantic abstraction computed by *GeneSyst* [BPS05] that projects the domain of the observed variables onto abstract domains.

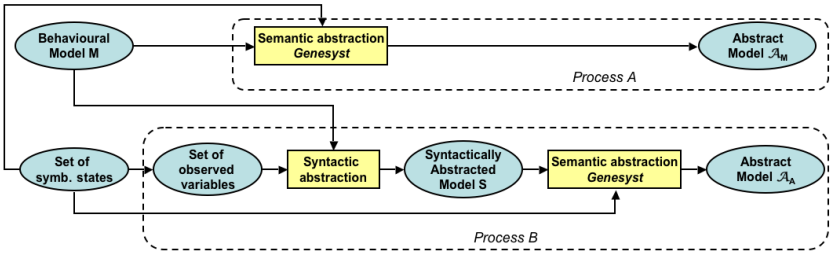


Fig. 4. Abstraction Process

5 Conclusion, Related Works and Further works

We have illustrated a method for abstracting an event system by elimination of some state variables. The abstraction is refined by the source model. It is useful for verifying properties and generating tests. The main advantage of our method is that it first performs syntactic transformations, which reduces the number of

POs generated and facilitates the proof of the remaining POs. This results in a gain of computation time. We believe that the bigger the ratio of the number of state variables to the number of observed variables is, the bigger the gain is. This conjecture needs to be confirmed by experiments on industrial size applications.

Many other works define model abstraction methods to verify properties. The methods of [GS97,BLO98,CU98] use theorem proving to compute the abstract model, which is defined over boolean variables that correspond to a set of *a priori* fixed predicates. In contrast, our method firstly introduces a syntactical abstraction computation from a set of observed variables, and further abstracts it by theorem proving. [CABN97] also performs a syntactic transformation, but requires the use of a constraint solver during a model checking process.

References

[BBJM09] F. Bouquet, P.-C. Bué, J. Julliand, and P.-A. Masson. Test generation based on abstraction and dynamic selection criteria. Research Report RR2009-02, Laboratoire d'Informatique de l'Université de Franche Comté, September 2009.

[BC00] D. Bert and F. Cave. Construction of Finite Labelled Transition Systems from B Abstract Systems. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[BJK+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. 2005.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[BPS05] D. Bert, M.-L. Potet, and N. Stouls. GeneSyst: a Tool to Reason about Behavioral Aspects of B Event Specifications. In *ZB'05*, volume 3455 of *LNCS*, 2005.

[CABN97] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[CU98] M.A. Colon and T.E. Uribe. Generating fnite-state abstractions of reactive systems using decision procedures. In *CAV'98*, volume 1427 of *LNCS*, 1998.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV'97*, volume 1254 of *LNCS*, 1997.

[JSBM09] J. Julliand, N. Stouls, P.-C. Bué, and P.-A. Masson. B model abstraction combining syntactic and semantics methods. Research Report RR2009-04, LIFC - Laboratoire d'Informatique de l'Université de Franche Comté, November 2009. 15 pages.

[LB08] M. Leuschel and M. Butler. ProB: An automated analysis toolset for the B method. *Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006.

Towards validation of requirements models

Atif Mashkoor[1] and Abderrahman Matoussi[2]

[1] LORIA – Nancy Université
Vandœuvre lès Nancy, France
{firstname.lastname}@loria.fr
[2] LACL – Université Paris-Est
Créteil Cedex, France
{firstname.lastname}@univ-paris12.fr

Abstract. The aim of this paper is to gradually introduce formalism in the requirement engineering phase in order to facilitate its validation. We analyze and elicit our requirements with KAOS, specify them into Event-B language, and then use the animation technique to rigourously validate the derived formal specification and consequently its semi-formal counterpart goal model against original customers' requirements.

1 Introduction

The use of formal methods for software development is escalating over the period of time. Most of the formal methods refine the initial mathematical model up to an extent where final refinement contains enough details for an implementation. The input to this formal specification phase is often the documents obtained during the requirements analysis activity which are either textual or semi-formal. Now there is a traceability gap between analysis and specification phases as verification of the semi-formal analysis model is difficult because of poor understandability of lower level of formalism of verification tools and validation of the formal specification is difficult for customers due to their inability to understand formal models.

Our objective is to bridge this gap by a gradual introduction of formalism into the requirement model in order to facilitate its validation. We analyse our requirements with KAOS (Knowledge Acquisition in autOmated Specification) [1] which is a goal-oriented methodology for requirements modeling, then we translate the KAOS goal model, following our derived precise semantics, into an Event-B [2] formal specification with the help of the platform RODIN[3], and finally we rigourously animate our specification with the help of the animator Brama [3], incorporated into platform RODIN, in order to validate the conformance of the specification to original requirements. With this approach we aim to reap benefits at two levels: customers can be involved into the development right form the start and consequently the requirement errors can be detected right on the spot.

[3] http://rodin-b-sharp.sourceforge.net

2 KAOS and Event-B

To analyze our requirements, we use KAOS which builds a data model in UML-like notation. The main KAOS goal defines an objective the system should meet, usually through the cooperation of multiple agents such as devices or humans, followed by several sub goals. Contrary to other requirements methods, such as i* [4], KAOS is well suited for our purpose because it can be extended with an extra step of formality which can fill in the gap between requirements and the later phases of development. The choice of Event-B as a formal specification language is due to its similarity and complementarity with KAOS. Firstly, Event-B is based on set theory with the ability to use standard first-order predicate logic facilitating the integration with the KAOS requirements model that is based on first-order temporal logic. Secondly, both Event-B and KAOS have the notion of refinement (constructive approach). Finally, KAOS and Event-B have the ability to model both the system and its environment.

3 Discussion

There are two main steps of our approach. First we translate our goal model into an Event-B specification with the help of our Event-B semantics, and later we animate the specification in order to validate that captured requirements are in accordance with original customer requirements. The whole process of validation is summed up by figure 1. Following is the brief elaboration of our approach:

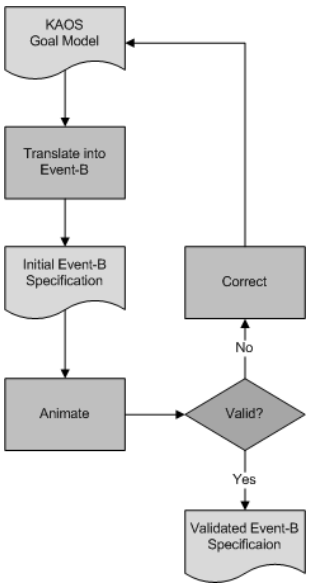


Fig. 1. The rigorous requirements validation process

3.1 The semantics step

The first step of our approach aims to express the KAOS goal model with Event-B by staying at the same level of abstraction which allows us to give this expression precise semantics. To achieve this objective, we use Event-B to formalize the KAOS refinement patterns that analysts use to generate a KAOS goal hierarchy. We primarily focus on most frequently used "Goal Patterns": the *Achieve goals*. The assertions in *Achieve goals* are expressed as following: $G\text{-}Guard \Rightarrow \diamond G\text{-}PostCond$, where *G-Guard* and *G-PostCond* are predicates. Symbol $\Rightarrow$ denotes the classical logical implication. Symbol $\diamond$ (the open diamond) represents the temporal operator "eventually" which ensures that a predicate must occur "at some time in future". Hence, such assertions demonstrate that from a state in which *G-Guard* holds, we can reach sooner or later to another state in which *G-PostCond* holds.

If we refer to the concepts of guard and postcondition that exist in Event-B, a KAOS goal can be considered as a postcondition of the system, since it means that a property must be established. The crux of our formalization is to express each KAOS goal as a B event, where the action represents the achievement of the goal. Then, we will use the Event-B refinement relation and additional custom-built proof obligations to derive all the subgoals of the system by means of B events. One may wonder whether the formalization of KAOS target predicates (i.e. the predicate after the diamond symbol) as B post conditions is adequate, since the execution of B events is not mandatory. At this very high level of abstraction, there is only one event for representing the parent goal. In accordance with the Event-B semantics, if the guard of the event is true, then the event necessarily occurs. For the new events built by refinement and associated to the subgoals, we guarantee by construction that no event prevent the post conditions to be established. For that, we have proposed an Event-B semantic for each KAOS refinement pattern by constructing set-theoretic mathematical models. This process continues until the complete specification of KAOS goal model into Event-B. A detailed discussion on this step can be found in [5]. Formalization of the goal patterns other than *Achieve goals* is a work in progress.

3.2 The animation step

Following the precise semantics discussed in previous section, we derive an initial Event-B specification of the KAOS goal model. The aim of this animation step is to validate this derived specification. Our approach to address this issue is based on following hypothesis: we presume if the animation of the specification reveals the same behavior that we intended while writing our goal model, then the Event-B specification would be considered as a valid formal representation of the customers' requirements. In order to achieve this, we execute our specification to check its behavior with the approach defined in [6]. We rigourously animate the specification at each refinement step. It not only indicates any deviation from original requirements right on the spot but also helps fixing the specification errors. If any deviation from the intended behavior is discovered, we go back to the source and rectify the error. The process continues until the specification fully adheres to the requirements.

4 Conclusion and future work

We present an approach to validate a semi-formal requirement model by the animation of its formal counterpart. We express a KAOS goal model capturing users' requirements into an Event-B specification language for a stepwise requirement validation process.

At theoretical level our approach seems promising as we have obtained some initial results at its both steps independently. However, we hope that our proposed combined approach of analysis, specification and validation is also feasible collectively. We aim to target at transportation domain [7] to test our hypothesis.

5 Acknowledgements

This work has been partially supported by the ANR project TACOS (Ref: ANR-06-SETI-017). We would also like to thank Jean-Pierre Jacquot and Régine Laleau for their valuable comments.

References

1. Van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications, Wiley (2009)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, Cambridge University Press (2009)
3. Servat, T.: BRAMA: A New Graphic Animation Tool for B Models, In: 7th International Conference of B Users (B'07), Springer-Verlag, Besançon, France (2007)
4. Yu, E.: Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering, In: 3rd IEEE International Symposium on Requirements Engineering (RE'97), IEEE, Annapolis, USA (1997)
5. Matoussi, A.: Expressing KAOS Goal Models with Event-B, In Doctoral Symposium of 16th International Symposium on Formal Methods (FM'09-DS), Eindhoven, The Netherlands (2009)
6. Mashkoor, A., Jacquot, J.P., Souquières, J.: Transformation Heuristics for Formal Requirements Validation by Animation, In: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert'09), York, UK (2009)
7. Mashkoor, A., Jacquot, J.P, Souquières, J.: B Événementiel pour la Modélisation du Domaine: Application au Transport, In: Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'09), Toulouse, France (2009)

B-ASM: Specification of ASM *à la* B

David Michel[1]*, Frédéric Gervais[2], Pierre Valarcher[2]

[1] LIX, CNRS, Polytechnique School, 91128 Palaiseau, France
dmichel@lix.polytechnique.fr
[2] LACL, Université Paris-Est
IUT Sénart Fontainebleau, Dpt. informatique, 77300 Fontainebleau, France
{frederic.gervais,pierre.valarcher}@univ-paris-est.fr

Abstract. We try to recover the proof correctness strength of the B method and the simplicity of the *Abstract State Machine* model (*ASM*) by constructing a B-ASM language. The language inherits from the language of substitution and from *ASM* program. The process of refinement leads us to a program expressed in the *ASM* syntax only. As each step of refinement is correct towards the specification, we obtain an *ASM* that is proved to be correct towards the specification.

Keywords. B Method, ASM, refinement, correctness, fixed point.

1 Introduction

This paper aims at extending the B language [1] in order to build ASM programs which are correct with respect to B-like logical specifications. On the one hand, the main strengths of the B formal method are: i) the ability to express logical statements, and ii) the construction of a correct implementation by refinement. On the other hand, from our viewpoint, the striking aspects of ASM are the non-bounded outer loop that can reach the fixed point of a program and the power to express naturally any kind of (sequential) algorithms.

This paper introduces a new specification language, called B-ASM, attempting to bridge the gap between these two languages, by taking advantage of the strengths of each approach. Our leitmotiv is to build an ASM which is correct with respect to a B-like specification. In that aim, we have extended the syntax and the semantics of B to take the non-bounded iteration into account. Moreover, the reuse of the well-founded theoretical relation of refinement from the B method is then straightforward. Rather than directly writing a complex ASM program, one can first specify the required logical properties of the program in a B-ASM specification. Then, we are able to build from the latter a correct ASM program, by proving the proof obligations (PO) associated to each refinement step. For instance, if we can determine a variant in the B-ASM specification for

* This author has been supported by the ANR-09-JCJC-0098-01 MaGiX project together with the Digiteo 2009-36 HD grant and région Île-de-France.

the outer loop, then the ASM program obtained by refinement is guaranteed to terminate.

In the following paragraphs, we briefly describe ASM and B.

Abstract State Machine. Abstract State Machines ($ASMs$) are known as a powerful theoretical tool to model (sequential) algorithms and this, at any level of abstraction of data structures ([7]). An ASM is a couple (A, π) where A is an algebra (that specifies data) and π is a program that operates over the algebra A. The program is a finite set of conditional rules testing (essentially) equalities over terms of the algebra and then updating the state in parallel (in the following we don't restrict the syntax of ASM to be in normal form, but we use a syntax closer to the Lipari Guide [6]).

The algebra A is initialized by an initial algebra and a computation is then the execution of π until a fixed point is reached or when a clashed-update occurs (a location is updated by two different values simultaneously).

This general model of computation has been used to model a large class of problems (see [8] for tools and general purpose and [9] for a large example).

More than a practical tool, the ASM model is an attempt to formalize the widely used notion of *algorithms*. The most important theoretical result is given in [7] and states that *any algorithm may be simulated step-by-step (in strict lock step) by an appropriate ASM.*

An Overview of B. B is a formal method [1] that supports a large segment of the software development life cycle: specification, refinement and implementation. In B, specifications are organized into abstract machines (similar to classes or modules). State variables are modified only by means of substitutions. The initialization and the operations are specified in a generalization of Dijkstra's guarded command notation, called the Generalized Substitution Language (GSL), that allows the definition of non-deterministic substitutions. For instance, in an abstract machine, we can define an operation with guarded substitutions, which are of the form **any** x **where** A **then** S **end**, where x is a variable, A a first-order predicate on x, and S a substitution. Such a substitution is non-deterministic because x can be any value that satisfies predicate A.

The abstract machine is then refined into concrete machines, by replacing non-deterministic substitutions with deterministic ones. At each refinement step, the operations are proven to satisfy their specification. Hence, through refinement steps and proofs, the final code is proven to be correct with respect to its specification. The B method is supported by several tools, like Atelier B [5], Click'n Prove [2] and the B-Toolkit [4].

Contribution. *Let M be a B-ASM machine, then we can construct an ASM which is correct with respect to M.*

This contribution is detailed in the next sections. Language B-ASM, the extension of B integrating ASM constructs, is presented in Sect. 2. Then, Sect. 3

provides a formal semantics for B-ASM, based on the weakest preconditions. Section 4 shows how to prove that an ASM program built by refinement is correct with respect to its B-ASM specification. Finally, Sect. 5 concludes the paper with some remarks and perspectives.

2 B-ASM Programs

ASM programs are decomposed in two parts:

- an initialization algebra, *i.e.* an initial state;
- a one-step transition function.

For executing an ASM program, the transition function is iteratively applied to the current state, starting from the initial state. The program stops when a fixed point is reached, in other words, the transition function does not alter the current state anymore.

We define B-ASM programs in the same way as in ASM programs, but the language used to define transition functions is enriched with operations akin to some non-deterministic B substitutions.

Definition 1. B-ASM *transition functions, i.e. B-ASM transitions, are defined by induction as follows:*

ASM operations:

- $f(\overrightarrow{t}) := u$ *is an B-ASM transition, where* $f(\overrightarrow{t})$ *and* u *are first-order terms;*
- **if** A **then** S **end** *is an B-ASM transition, where* A *is a formula and* S *is an B-ASM transition;*
- **par** $\overrightarrow{S}$ **end** *is an B-ASM transition, where* $\overrightarrow{S}$ *is a list of B-ASM transition;*
- **skip** *is an B-ASM transition;*

non-deterministic operations:

- $f(\overrightarrow{t}) :\in E$ *is an B-ASM transition, where* $f(\overrightarrow{t})$ *is a first-order term and* E *a set;*
- **@**x**.**S *is an B-ASM transition, where* x *is a variable and* S *is an B-ASM transition;*
- **choice** S **or** T **end** *is an B-ASM transition, where* S *and* T *are B-ASM transition;*
- **any** x **where** A **then** S **end** *is an B-ASM transition, where* x *is a variable,* A *a formula and* S *an B-ASM transition.*

Let us now focus on the B specification language for the purposes of this paper. In B, each abstract machine encapsulates state variables (introduced by keyword **VARIABLES**), an invariant typing the state variables (in **INVARIANT**), an initialization of all the state variables (**INITIALISATION**), and operations on the state variables (**OPERATIONS**). The invariant is a first-order predicate in a simplified version of the ZF-set theory, enriched by many relational operators.

We define the B-ASM specification language as a simple modification of the B language, in order to specify ASM programs. In B-ASM, the vocabulary of algebras (*i.e.* states) is introduced by keyword **VARIABLES**, the variables are typed in the clause **INVARIANT**, the **INITIALISATION** clause contains the definition of the initial states as parallel (non-deterministic) substitutions, and the ASM transition is described in the clause **OPERATION**. In this paper, we deal with terminating programs, so we add to the syntax an additional clause called **VARIANT**. The latter defines the integer value which strictly decreases at each iteration step.

To illustrate the B-ASM approach, we consider a machine specifying the maximum of an array of integer values.

MACHINE $Maximum(tab)$
CONSTRAINTS $tab \in seq_1(\mathbb{N})$
VARIABLES $maxi$ /* Vocabulary */
INVARIANT $maxi \in \mathsf{ran}(tab)$ /* Typing */
CONSTANTS $Maxi$
PROPERTIES
$Maxi \in seq_1(\mathbb{N}) \rightarrow \mathbb{N} \ \wedge$
$\forall t \in seq_1(\mathbb{N}).(Maxi(t) \in \mathsf{ran}(t) \ \wedge \ \forall e \in \mathsf{ran}(t).(e \leq Maxi(t)))$
INITIALISATION $maxi := tab(1)$ /* Initial state */
VARIANT $Maxi(tab) - maxi$ /* Halting condition */
OPERATION /* B-ASM transition function */
$maxi := Maxi(tab)$
END

In this abstract machine, we only specify the logical properties of the expected results, without defining an algorithm to compute them. At this stage, the OPERATION clause consists of a non-deterministic B-ASM transition. In order to obtain a formalized ASM (*i.e.* without non-deterministic operations), this abstract machine has to be *refined* into a deterministic specification. Our approach consists in adapting the B refinement relation to the B-ASM method. Since the semantics of the B language is based on weakest preconditions (WP), we have to provide the WP semantics of the B-ASM transition language defined in Def. 1. This semantics will be presented in Sect. 3. In our example, the following machine is one of the possible refinements of abstract machine $Maximum(tab)$.

REFINEMENT $MaximumASM(tab)$
REFINES $Maximum(tab)$
VARIABLES $length, i, maxi$ /* Vocabulary */
INVARIANT /* Typing */
$length = size(tab) \ \wedge$
$i \in 1..length \ \wedge$
$maxi = Maxi(tab \uparrow i)$
INITIALISATION /* Initial states */
$length := size(tab);$

```
    i := 1;
    maxi := tab(1)
VARIANT length − i                    /* Halting condition */
OPERATION                             /* ASM transition function */
    if i < length then
        par
            i := i + 1
            if tab(i + 1) > maxi then
                maxi := tab(i + 1)
            end
        end
    end
END
```

Observe that we have formalized an ASM, since there are no non-deterministic operations used in the OPERATION clause. Intuitively, the algorithm specified by the ASM computes the maximum of array *tab*. By using the refinement relation inspired from B, we can prove that this ASM faithfully implements its abstract specification.

3 Weakest Precondition Semantics of Transitions

To define the weakest precondition semantics, we first introduce a function m which associates an integer to each list of B-ASM transitions. It will be used to define the semantics by induction.

Definition 2. *For all B-ASM programs S, we define an integer $m(S)$ by induction on S, and for all lists $\overrightarrow{S}$ of B-ASM programs, we will write $m(\overrightarrow{S})$ for $\sum_{S\in\overrightarrow{S}} m(S)$:*

- $m(f(\overrightarrow{t}) := u) = 0$;
- $m(\textbf{if } A \textbf{ then } S \textbf{ end}) = 1 + m(S)$;
- $m(\textbf{par } \overrightarrow{S} \textbf{ end}) = 1 + m(\overrightarrow{S})$;
- $m(\textbf{skip}) = 1$;
- $m(f(\overrightarrow{t}) :\in E) = 1$;
- $m(@x.S) = 1 + m(S)$;
- $m(\textbf{choice } S \textbf{ or } T \textbf{ end}) = 1 + m(S) + m(T)$;
- $m(\textbf{any } x \textbf{ where } A \textbf{ then } S \textbf{ end}) = 1 + m(S)$.

We now define the weakest precondition predicate for each list of B-ASM transitions. Lists are here used to take the parallel B-ASM transitions into account.

Definition 3. *Let $\overrightarrow{f(\overrightarrow{t}) := u}$ denote the list of substitutions $(f_i(\overrightarrow{t_i}) := u_i)_{0\le i\le n}$. For all lists $\overrightarrow{S}$ of B-ASM programs, we define the formula $[\overrightarrow{S}]P$ by induction on $m(\overrightarrow{S})$; let $\overrightarrow{Z}$ be the list $\overrightarrow{f(\overrightarrow{t}) := u}$; we consider all cases according to definition 1:*

- $[\overrightarrow{Z}]P = \begin{cases} P(\overrightarrow{f \backslash f \Leftarrow \{\overrightarrow{t} \mapsto u\}}) & \text{if the substitution is consistent;} \\ P & \text{if not;} \end{cases}$
- $[\overrightarrow{Z}, \textbf{if } A \textbf{ then } S \textbf{ end}, \overrightarrow{T}]P = (A \Rightarrow [\overrightarrow{Z}, S, \overrightarrow{T}]P) \wedge (\neg A \Rightarrow [\overrightarrow{Z}, \overrightarrow{T}]P)$;
- $[\overrightarrow{Z}, \textbf{par } \overrightarrow{S} \textbf{ end}, \overrightarrow{T}]P = [\overrightarrow{Z}, \overrightarrow{S}, \overrightarrow{T}]P$;
- $[\overrightarrow{Z}, \textbf{skip}, \overrightarrow{T}]P = [\overrightarrow{Z}, \overrightarrow{T}]P$;
- $[\overrightarrow{Z}, f(\overrightarrow{t}) :\in E, \overrightarrow{T}]P = \forall x. x \in E \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, \overrightarrow{T}]P$;
- $[\overrightarrow{Z}, @x.S, \overrightarrow{T}]P = \forall x. [\overrightarrow{Z}, S, \overrightarrow{T}]P$;
- $[\overrightarrow{Z}, \textbf{choice } S \textbf{ or } T \textbf{ end}, \overrightarrow{U}]P = [\overrightarrow{Z}, S, \overrightarrow{U}]P \wedge [\overrightarrow{Z}, T, \overrightarrow{U}]P$;
- $[\overrightarrow{Z}, \textbf{any } x \textbf{ where } A \textbf{ then } S \textbf{ end}, \overrightarrow{T}]P = \forall x. A \Rightarrow [\overrightarrow{Z}, S, \overrightarrow{T}]P$.

The following theorem allows us to prove that the semantics of parallel B-ASM transitions does not depend on the order of these transitions. Indeed, if we alter the order of transitions in a list, the resulting formulas are syntactically different. However, the theorem states that these formulas are semantically equivalent.

Theorem 1. *For all lists $\overrightarrow{S}$ of B-ASM programs and for all permutations σ, formula $[\overrightarrow{S}]P$ is equivalent to formula $[\sigma(\overrightarrow{S})]P$.*

Proof. We proceed by induction on $m(\overrightarrow{S})$; we remark that for all permutations $\overrightarrow{S'}$ of $\overrightarrow{S}$ we have $m(\overrightarrow{S}) = m(\overrightarrow{S'})$. Let us write $\overrightarrow{S} = \overrightarrow{Z}, T, \overrightarrow{U}$ where $\overrightarrow{Z} = \overrightarrow{f(\overrightarrow{t}) := u}$ and $T \neq f(\overrightarrow{t}) := u$; in the same way, we write $\overrightarrow{S'} = \overrightarrow{Z'}, T', \overrightarrow{U'}$ where $\overrightarrow{Z'} = \overrightarrow{f'(\overrightarrow{t'}) := u'}$ and $T' \neq f'(\overrightarrow{t'}) := u'$. We consider as an example $T = \textbf{if } A \textbf{ then } V \textbf{ end}$ and $T' = f(\overrightarrow{t}) :\in E$; since formulas of the form $[\overrightarrow{S}]P$ are in positive occurrences in definition 3, the other cases are quite similar. By definition we have:

$$[\overrightarrow{S}]P = (A \Rightarrow [\overrightarrow{Z}, V, \overrightarrow{U}]P) \wedge (\neg A \Rightarrow [\overrightarrow{Z}, \overrightarrow{U}]P)$$

There is a permutation μ such that $\mu(\overrightarrow{Z}, V, \overrightarrow{U}) = \overrightarrow{Z}, T', V, \overrightarrow{U''}$; by induction hypothesis, $[\overrightarrow{Z}, V, \overrightarrow{U}]P$ is equivalent to $[\overrightarrow{Z}, T', V, \overrightarrow{U''}]P$. In the same way, $[\overrightarrow{Z}, \overrightarrow{U}]P$ is equivalent to $[\overrightarrow{Z}, T', \overrightarrow{U''}]P$. Thus, we have:

$$[\overrightarrow{S}]P \Leftrightarrow (A \Rightarrow [\overrightarrow{Z}, T', V, \overrightarrow{U''}]P) \wedge (\neg A \Rightarrow [\overrightarrow{Z}, T', V, \overrightarrow{U''}]P)$$

By definition, we have:

$$[\overrightarrow{Z}, T', V, \overrightarrow{U''}]P = \forall x. x \in E \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, V, \overrightarrow{U''}]P$$

and:

$$[\overrightarrow{Z}, T', \overrightarrow{U''}]P = \forall x. x \in E \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P$$

Thus, according to usual boolean tautologies, we have:

$$[\overrightarrow{S}]P \Leftrightarrow \forall x. x \in E \Rightarrow (A \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, V, \overrightarrow{U''}]P) \wedge (\neg A \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P)$$

By induction hypothesis, $[\overrightarrow{Z}, f(\overrightarrow{t}) := x, V, \overrightarrow{U''}]P$ is equivalent to $[\overrightarrow{Z}, V, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P$; thus, we have:

$$[\overrightarrow{S}]P \Leftrightarrow \forall x.x \in E \Rightarrow [\overrightarrow{Z}, T, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P$$

There is a permutation ρ such that $\rho(\overrightarrow{Z}, T, f(\overrightarrow{t}) := x, \overrightarrow{U''}) = \overrightarrow{Z'}, f(\overrightarrow{t}) := x, \overrightarrow{U'}$; by induction hypothesis, $[\overrightarrow{Z}, T, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P$ is equivalent to $[\overrightarrow{Z'}, f(\overrightarrow{t}) := x, \overrightarrow{U'}]P$. Hence we have $[\overrightarrow{S}]P \Leftrightarrow [\overrightarrow{Z'}, T', \overrightarrow{U'}]P$; thus $[\overrightarrow{S}]P$ is equivalent to $[\sigma(\overrightarrow{S})]P$.

4 Refinement of Programs

In Def. 3, we have defined the semantics for B-ASM transitions. Now, we have to define the semantics for the associated program. The latter has the same semantics as the program of the following form:

while $\overrightarrow{f' \neq f}$ **do**
  $\overrightarrow{f' := f}$;
  S;
  if $\overrightarrow{f' = f}$ **then** $terminate := 0$ **end**
  invariant I
  variant $V + terminate$
end

In this program, we have introduced several notations:

- $\overrightarrow{f}$ denotes the list of variables;
- $\overrightarrow{f'}$ is a list of fresh variables which are used to save the values of $\overrightarrow{f}$ from the previous state; they are of the same type as $\overrightarrow{f}$ augmented with special values that denote undefinedness. For instance, in the machine $MaximumASM(tab)$, some clauses are implicitly extended:
  - variables $length'$, i', $maxi'$ and $terminate$ are added to clause VARIABLES;
  - the following formulas are in clause INVARIANT:
    $$length' = size(tab) \cup \{\bot\} \ \wedge$$
    $$i' \in 1..length' \cup \{\bot\} \ \wedge$$
    $$maxi' = Maxi(tab \uparrow i') \cup \{\bot\} \ \wedge$$
    $$terminate \in \{0, 1\}$$
  - in clause INITIALISATION, $length'$, i', and $maxi'$ are initialized to $\bot$, and $terminate$ is initialized to 1;
- I denotes the body of the INVARIANT clause;
- V denotes the body of the VARIANT clause;
- S denotes the body of the OPERATION clause.

The proof obligation associated to the above-mentioned program is derived from the classical proof obligations associated to WHILE substitutions in the B method. Let us write B the loop body:

$$B \triangleq \overrightarrow{f' := f}\ ;\ S\ ;\ \textbf{if}\ \overrightarrow{f' = f}\ \textbf{then}\ terminate := 0\ \textbf{end}$$

In order to prove that the program establishes predicate P, we have to prove that:

1. the loop body preserves the invariant:

$$I\ \wedge\ \overrightarrow{f' \neq f} \Rightarrow [B]I \quad (PO1)$$

2. the variant is well-typed:

$$I \Rightarrow V + terminate \in \mathbb{N} \quad (PO2)$$

3. the variant strictly decreases at each iteration step:

$$I\ \wedge\ \overrightarrow{f' \neq f} \Rightarrow [n := V + terminate][B](V + terminate < n) \quad (PO3)$$

4. when the loop terminates, the program establishes predicate P:

$$I\ \wedge\ \overrightarrow{f' = f} \Rightarrow P \quad (PO4)$$

Refinement Proof. The proof obligations (PO) associated to refinement are of the following form:

1. $[Init']\neg[Init]\neg J$ (PO init)
2. $I \wedge J \Rightarrow [Subst']\neg[Subst]\neg J$ (PO op)

where $Init$ represents the initialization substitutions, $Subst$ the operation substitutions, and I the invariant in the abstract machine. $Init'$, $Subst'$, and J denote the counterparts of $Init$, $Subst$, and I, respectively, in the refinement machine. The use of negation allows non-determinism to be taken into account. These two POs guarantee that the execution of the concrete initialization (the concrete operation, respectively) is not in contradiction with the effects of the abstract initialization (the asbtract operation, resp.).

For instance, in our example dealing with the maximum of an array of integer values, the proof of (PO init) is straightforward. Double negation is not needed, because no substitution is non-deterministic in this example. Since B-ASM programs mainly consist of a WHILE loop, (PO op) requires the decomposition of $[Subst']$ into the four above-mentioned POs associated to programs.

Proof obligation $(PO1)$ can be proved by a case analysis. Either the new element in tab is a new maximum, in that case, invariant $maxi = Maxi(tab \uparrow i)$ is preserved by substitution $maxi := tab(i + 1)$, or the new element is not a maximum, consequently the invariant is also preserved.

Proof obligation $(PO2)$ is straightforward.

For $(PO3)$, the proof consists in applying $i := i+1$ at each step of iteration; hence, the variant strictly decreases. Variable *terminate* allows us to decrease the variant even in the last iteration step, when $i = length$, but just before $\overrightarrow{f' = f}$.

In this example, predicate P in $(PO4)$ is:

$$[maxi := Maxi(tab)](length = size(tab) \ \wedge \ i \in 1..length \\ \wedge \ maxi = Maxi(tab \uparrow i) \ )$$

The latter can be rewritten into:

$$length = size(tab) \ \wedge \ i \in 1..length \ \wedge \ Maxi(tab) = Maxi(tab \uparrow i)$$

(PO4) is straightforward, since at each iteration step, we guarantee by the invariant clause that $Maxi$ restricted to the i first elements is effectively the maximum. Once all the elements are analysed, $Maxi(tab) = Maxi(tab \uparrow length)$.

5 Conclusion

The B method and the *Abstract State Machine* Êmodel have their own strength: proof correctness during the software development life cycle for the B method and algorithmic completeness for ASM model.

By mixing the two models, we expect to conserve both the qualities of the B method and the usability of $ASMs$. For this, we add the $ASMs$ syntax to the B language of substitution and give a semantics for the weakest precondition and a semantics for the program obtained by refinement.

At the end of the process a new B_0 program is obtained following strictly the syntax of a π program of an ASM, moreover the process has followed the proof correctness of B method refinement.

The challenge is now, to verify the efficiency of the new method in a real case study and of course, to develop tools.

References

1. J.R. Abrial. *The B-Book: Assigning programs to meanings.* CUP, 1996.
2. Abrial, J.R., Cansell, D.: Click'n Prove: Interactive proofs within set theory. In *TPHOLs 2003, Rome, Italy*, LNCS 2758, Springer-Verlag, September 2003.
3. J.R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B'98*, volume 1393 of *LNCS*, pages 83–128, Montpellier, France, April 1998. Springer-Verlag.
4. B-Core (UK) Ltd.: B-Toolkit, `http://www.b-core.com/btoolkit.html`
5. Clearsy: Atelier B, `http://www.atelierb-societe.com`
6. Yuri Gurevich, *Evolving Algebras 1993: Lipari Guide.* Specification and Validation Methods, 1993, Oxford University Press.
7. Yuri Gurevich, *Sequential Abstract State Machines capture Sequential Algorithms.* ACM Transactions on Computational Logic, 1(1) (July 2000), 77-111.
8. E. Boerger and R. Staerk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.
9. R. Staerk, J. Schmid and E. Boerger, *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag, 2001.

Introducing Specification-based Data Structure Repair Using Alloy (Informal Extended Abstract)

Razieh Nokhbeh Zaeem and Sarfraz Khurshid

University of Texas, Austin TX 78712, USA
{rnokhbehzaeem,khurshid}@ece.utexas.edu

Abstract. Approaches that use specifications, e.g., assertions, to detect erroneous program states are common. We have developed a novel specification-based approach for data structure repair, which allows repairing erroneous executions in deployed software. The key novelty is our support for rich behavioral specifications, such as those that relate pre-states with post-states to accurately specify expected behavior and hence to enable precise repair.

1 Introduction

As software failures have become expensive and frequent, the need for creating new methodologies that deliver reliable software at a lower cost has become urgent. Much of the existing research effort is devoted to requirements, architecture, design, implementation and testing, activities that are performed before the deployment of a software system. In contrast, little effort is devoted to developing methodologies that handle errors that arise during system executions after the deployment.

While several different techniques utilize specifications to check correctness of programs before they are deployed, the use of specifications in deployed software is more limited, largely taking the form of runtime checking where assertions form a basis for detecting erroneous program states and terminating erroneous executions in failures. The standard approach when an erroneous program state is detected at runtime, say due to an assertion violation, is to terminate the program, debug it if possible, and re-execute it. While this halt-on-error approach is useful for eliminating transient errors or for debugging purposes, it does not present a feasible solution for deployed software that is faulty and cannot be promptly debugged or re-deployed.

Recent work introduced constraint-based repair where data structure constraints written using first-order logic [1] or as Java assertions [2] are used as a basis for repairing erroneous states. However, data structure constraints are too weak a form of specification for error recovery in general. To illustrate, in object-oriented programs, the *class invariant* (which defines the data structure constraints for the valid objects of the class) applies to the entry and exit points of all public methods—even though the precise behaviors of the methods may be very different. For example, consider an erroneous implementation of a method to insert an element into a *binary tree*—an acyclic data structure. Previous approaches [1, 2] to constraint-based repair would accept an empty tree as a valid structure since it satisfies the acyclicity constraint. However, an empty tree is unlikely to be a valid output of insert.

We have developed a specification-based approach for data structure repair, which allows repairing erroneous executions in deployed software by repairing erroneous

```
class LinkedList {
    Node header;
    int size;  // number of nodes

    static class Node {
        Node next;
        int elt; }

    void remove(int x) { // erroneous implementation
        // should remove all nodes that have element x
        if (header == null) return;
        Node pointer = header.next;
        Node prevPointer = header;
        while (pointer != null) {
            if (pointer.elt == x) {
                 prevPointer.next = pointer.next;
                 pointer = pointer.next;
                 size--; }
            else {
                 prevPointer = prevPointer.next;
                 pointer = pointer.next; } }
        if (header.elt == x) {
            // the next line should be un-commented
            // header = header.next;
            // the next line is incorrect; it should be "size--;"
            size++; } } }
```

Fig. 1. Singly-linked list in Java. An erroneous `remove` method.

```
pred repOk(l: LinkedList){ // class invariant of LinkedList
   all n:l.header.*next | n !in n.^next // acyclicity
   # l.header.*next = int l.size // size ok
   all n, m: l.header.*next | int m.elt = int n.elt => n = m // unique elements }

pred remove_postcondition(This: LinkedList, x: Int){
   repOk[This]
   This.header.*next.elt - x = This.header`.*next`.elt` }
```

Fig. 2. Class invariant for `LinkedList` and post-condition for `remove` in Alloy.

states. The key novelty is the support for rich behavioral specifications, such as those that relate pre-states with post-states to accurately specify expected behavior and hence to enable precise repair. This informal extended abstract gives an example to illustrate the repair problem (Section 2), defines the problem and states the basic ideas behind our approach (Section 3), and concludes with future work (Section 4).

2 Example

To illustrate specification-based repair, consider a singly-linked list data structure (Figure 1). Each list object has a `header` node and a `size` field that caches the number of nodes in the list. Each node has a `next` pointer and contains an integer element (`elt`). The method `remove` removes all occurrences of the given integer (`x`) from the given list (`this`). The user provides the class invariant for `LinkedList` and the post-condition for `remove` in Alloy (Figure 2), which is used as a basis for repairing erroneous outputs. The class invariant requires that the list should be acyclic, contain unique integer elements, and have correct size. The post-condition additionally requires that the output should not include the element to remove. Back-tick (‘ ’) is syntactic sugar to represent post-state [4]. Note how the post-condition relates the set of list elements in the post-state with those in the pre-state to precisely specify correctness of remove.

To contrast with previous work [1,2], if we only use the class invariant (`repOk`) as a basis of repair, the repaired output list could be any correct instance of singly linked list, irrespective of its relationship with the pre-state. Our approach repairs faulty outputs of the remove method including those that violate the class invariant as well as those that fail to satisfy the pre- and post-condition relation.

3 Specification-based Data Structure Repair

We address the following repair problem: Let ϕ be a method postcondition that relates pre- and post-states such that $\phi(r, t)$ if and only if pre-state r and post-state t satisfy the post-condition. Given a valid pre-state u, and an invalid post-state s (i.e., $!\phi(u, s)$), mutate s into state s' such that $\phi(u, s')$.

Our approach is based on the view of a specification as a non-deterministic implementation, which may permit a high degree of non-determinism. The Alloy tool-set [3] provides the enabling technology for writing specifications and systematically repairing erroneous states. One initial technique that we developed is to transform the repair problem into a constraint solving problem and leverage the Alloy tool-set as a solving machine, ignoring the erroneous state. Although this technique provides a correct output, it might be infeasible for larger states. Our key insight to improve this technique is to use any correct state mutations by an otherwise erroneous execution to prune the non-determinism in the specification, thereby transmuting the specification to an implementation that does not incur a prohibitively high performance penalty. Moreover, using the faulty post-state as the start point of the repair process avoids unnecessary perturbations during the repair process. We are working on extensions of this idea to build an effective and efficient repair framework that supports rich behavioral specifications.

4 Conclusion and Future Work

We introduced a novel use of rich behavioral specifications for systematic data structure repair using the Alloy tool-set as an enabling technology. Our initial technique to perform repair was to use the Alloy tool-set to solve the post condition independently of the erroneous state. However, erroneous states likely contain valuable information about expected outputs and can serve as the basis of the repair. We are developing new algorithms that perform repair leveraging the currently erroneous state to prevent unnecessary perturbation in the data structure and improve repair accuracy and performance.

Acknowledgment

This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967, CCF-0702680, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

1. B. Demsky. *Data Structure Repair Using Goal-Directed Reasoning*. PhD thesis, Massachusetts Institute of Technology, Jan. 2006.
2. B. Elkarablieh. *Assertion-based Repair of Complex Data Structures*. PhD thesis, University of Texas at Austin, 2009.
3. D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, 2006.
4. D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.

On the modelling and analysis of Amazon Web Services access policies

David Power, Mark Slaymaker, and Andrew Simpson

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD
United Kingdom

Abstract. Cloud computing is a conceptual paradigm that is receiving a great deal of interest from a variety of major commercial organisations. By building systems which run within cloud computing infrastructures, problems related to scalability and availability can be reduced, and, from the point of view of consumers of such infrastructures, abstracted away from. As such infrastructures tend to be shared, it is important that access to the sub-components of each system is controlled. One of the first languages for controlling access to services within a cloud is the Amazon Web Services access policy language. In this paper we present two formal models of this language—one in Z and one in Alloy—and show how the Alloy model might be used to test properties of multiple policies and to generate and test candidate policies.

1 Introduction

Cloud computing (see, for example, [1]) is a conceptual paradigm that is receiving a great deal of interest from a variety of major commercial organisations. By building systems which run within cloud computing infrastructures, problems related to scalability and availability can be reduced, and, from the point of view of consumers of such infrastructures, abstracted away from. As such infrastructures tend to be shared, it is important that access to the sub-components of each system is controlled.

Many cloud computing infrastructures have emerged over the past few years; at the time of writing, Amazon Web Services (AWS) [2] is one of the most widely used. AWS consists of a number of different components, which can be used in combination or alone. One common usage model is to use Elastic Compute Cloud (EC2) instances to process information and to use the Simple Queue Service (SQS) [3] to handle requests and responses. For example, a language translation service might involve an end-user initially submitting input to a web page. Then the inputted string would form part of a request placed in a queue (the request queue). The EC2 instance would consume messages from the queue, perform the required task (in this case, translation), and then put a message containing the result in a second queue (the response queue). The result will then be consumed by the web site.

If all of the sub-components of a system use the same security credentials, it is possible to restrict access using an 'all-or-nothing' approach. However, there are situations where more complex controls are appropriate. For example, there may be a key requirement that a particular service is only available during certain time periods. For this reason, the AWS access policy language was introduced, which enables access to be restricted based on a number of factors, including the time of the request and the originating IP address, as well as more common factors, such as the action that is being performed and the resource that is being acted upon. Currently, only the SQS service supports the AWS access policy language, but there are plans for it to be used on additional components.

As the complexity of access control policies increases, there is a corresponding increase in the risk that a mistake might be made when defining these policies. The value of analysing such policies has been demonstrated by the work of others in the community, such as Ryan and colleagues (see, for example, [4]) and Bryans and Fitzgerald (see, for example, [5])).

In this paper we seek to reduce that risk with the appropriate application of formal methods. We use a hybrid approach of using both the Z specification language [6] and the Alloy modelling language [7]. Each language has its strengths and weaknesses, with Z better suited to formal proof and Alloy being better suited to automatic analysis. The differences in the languages reflect the intentions of their creators with each having its place. In this paper the Z model is used as a starting point for the Alloy model, which we use for the examples presented. From a pragmatic perspective, our choice of leveraging both languages comes down to the fact that there may be some circumstances in which a fully formal proof is necessary; typically, however, the excellent support for model finding offered by Alloy will be appropriate.

In Section 2 we provide a necessarily brief description of the AWS access policy language. In Section 3 we present a model of the AWS policy language written in the Z specification language. In Section 4 we translate (manually) the Z model into the Alloy language and also extend the model to support the specifics of policies written for the SQS service. We give two examples of the use of the Alloy model in Section 5. In the first example we look at a scenario where multiple queues are used as part of a simple system; we then use the Alloy Analyzer to find examples of situations in which the system will fail to perform as required. In the second example we look at the use of the Alloy Analyzer to assist in the creation of policies that meet a set of complex requirements. By creating a set of requests with known outcomes we can generate candidate policies. Finally, in Section 6, we summarise the contribution and explore avenues of potential future work.

2 Background

The AWS access policy language allows one to construct policies that have a tree-like structure, consisting of sub-components, each of which may give an independent result—with these results being combined systematically to arrive

at a final decision. In this respect, it has similarities with the OASIS standard XACML (eXtensible Access Control Markup Language) [8].

The AWS access policy language makes permit or deny decisions based on the identity of the user, the action they are trying to perform, and the resource they are trying to act on. Users are identified by a *principal*, which is used as part of the authentication process and is tied to a specific AWS account. In addition, a number of environmental factors may be used as part of the decision making process, these are identified by the use of keys.

Each policy consists of a number of statements which contain a description of the requests they apply to, plus an effect, which may be permit or deny. Each statement contains lists of actions, lists of resources and lists principals, plus a number of conditions which must be met. The conditions are related to the key values of the request.

If multiple statements match a request, then deny effects take precedence over permit effects. If no statements match, then the effect is referred to as a soft deny: that is to say that final effect will be deny unless another policy has an effect of permit.

Conditions have a type, which is a matching relation such as string equality or 'before' on date-time values. Conditions also contain a number of clauses, all of which must hold for the condition to be met. Each clause consists of a key and a number of values: if any of the values match the request's key value then the clause holds.

When writing a policy for use with the SQS service a number of additional restrictions apply. Each policy may only contain statements relating to a single queue, the identity of which is used as the resource value. Only the following actions may be used: `ReceiveMessage`; `SendMessage`; `DeleteMessage`; `ChangeMessageVisibility`; and `GetQueueAttributes`.

The only available keys are the standard keys, which are: `CurrentTime (DateTime)`; `SecureTransport (Boolean)`; `SourceIP (IP Address)`; and also `UserAgent (String)`.

The available condition types are dependent on the types of the available keys, and include `DateEquals`, `DateLessThan`, `IpAddress`, and `StringEquals`.

In addition, there is also a restriction that all policies and statements are uniquely identified.

The following example (from [3]) illustrates a policy in which all users are given `ReceiveMessage` permission for the queue named `987654321098/queue1`, but only between noon and 3:00 p.m. on January 31, 2009.

```
{
  "Version": "2008-10-17",
  "Id": "Queue1_Policy_UUID",
  "Statement":
    {
       "Sid":"Queue1_AnonymousAccess_ReceiveMessage_TimeLimit",
       "Effect": "Allow",
       "Principal": {
          "AWS": "*"
       },
       "Action": "SQS:ReceiveMessage",
       "Resource": "/987654321098/queue1",
       "Condition" : {
```

```
            "DateGreaterThan" : {
               "AWS:CurrentTime":"2009-01-31T12:00Z"
            },
            "DateLessThan" : {
               "AWS:CurrentTime":"2009-01-31T15:00Z"
            }
         }
      }
}
```

One of the benefits of this language is that, when compared to, for example, XACML, it is relatively streamlined: making a mapping to a formal representation more straightforward than would typically be the case.

3 A Z representation

In this section we describe the structure of an AWS access policy using the Z specification language. The model represents all aspects of the policy language, but does not include any explicit value types. To ease readability, some types are used before they are defined.

Each policy has a unique identifier, a policy language version number (which currently has no practical impact upon policies or their evaluation, but is included for the sake of completeness) and a non-empty list of statements. As the order of statements has no effect on request evaluation, the list is represented as a set. There is currently only one version of the policy language, represented by the constant $v1$.

$$[PolicyID]$$

$$Version ::= v1$$

Policy

$$version : Version$$
$$id : PolicyID$$
$$statements : \mathbb{P}_1\ Statement$$

Each statement places conditions upon the requests to which it applies: if the statement applies, it may have an effect which can be to either allow or deny access. During evaluation, a deny decision may be be the result of a statement with an effect of deny—called a *hard deny*—or the absence of an applicable statement with an effect of allow—called a *soft deny*. The absence of an effect in a statement is modelled as a soft deny. In our model, we have explicitly captured the soft deny effect; there is no need to model the hard deny effect explicitly.

Similarly to a policy, each statement has a unique identifier. It also details the principals, actions and resources to which it applies. The principals represent the identity of the requester, the actions represent the action to be performed, and the resource represent the entity that the action will be performed on. Each of these can take multiple values and are represented by non-empty sets.

Finally, a statement may contain a number of conditions which further restrict the applicability of a statement.

$[StatementID, Principal, Action, Resource]$

$Effect ::= Allow \mid Deny \mid SoftDeny$

Statement

$sid : StatementID$
$effect : Effect$
$principals : \mathbb{P}_1\ Principal$
$actions : \mathbb{P}_1\ Action$
$resources : \mathbb{P}_1\ Resource$
$conditions : \mathbb{P}\ Condition$

Conditions consist of a matching relation, referred to as the type of the condition, and a number of clauses describing the values to be matched. Each clause contains a key and a number of values. Each key represents a specific piece of data about the request, such as the current time or the IP address from which the request originated. These are compared with concrete values using the matching relation corresponding to the type of the condition.

Condition

$type : CondType$
$clauses : \mathbb{P}_1\ Clause$

$[Key, Value]$

Clause

$key : Key$
$values : \mathbb{P}_1\ Value$

CondType

$match : Value \leftrightarrow Value$

As well as having a model of the policy, it is also necessary to model a request. Each request contains a single principal, action and resource. In addition, the *keys* function provide the value associated with each key. It is assumed that the *keys* function is total.

Request

$principal : Principal$
$action : Action$
$resource : Resource$
$keys : Key \rightarrow Value$

At the top level, a request is evaluated against a set of policies: if any policy evaluates to *Deny* or no policy evaluates to *Allow*, then the result is a *Deny*. If there are no *Deny*s and at least one *Allow*, then the result is *Allow*.

$$
\begin{array}{|l}
EvalPolicies : Request \rightarrow (\mathbb{P}\, Policy) \rightarrow \{Allow, Deny\} \\
\hline
\forall\, r : Request \bullet EvalPolicies(r) = \\
\quad (\lambda\, pols : \mathbb{P}\, Policy \bullet Deny) \\
\quad \oplus \\
\quad (\lambda\, pols : \mathbb{P}\, Policy \mid \\
\qquad (\exists\, p : pols \bullet EvalPolicy(r)(p) = Allow) \bullet Allow) \\
\quad \oplus \\
\quad (\lambda\, pols : \mathbb{P}\, Policy \mid \\
\qquad (\exists\, p : pols \bullet EvalPolicy(r)(p) = Deny) \bullet Deny)
\end{array}
$$

The evaluation of a policy is dependent on the evaluation of the statements it contains. When no statement results in a *Deny* or *Allow*, the result is a *SoftDeny*.

$$
\begin{array}{|l}
EvalPolicy : Request \rightarrow Policy \rightarrow Effect \\
\hline
\forall\, r : Request \bullet EvalPolicy(r) = \\
\quad (\lambda\, p : Policy \bullet SoftDeny) \\
\quad \oplus \\
\quad (\lambda\, p : Policy \mid \\
\qquad (\exists\, s : p.statements \bullet EvalStatement(r)(s) = Allow) \bullet Allow) \\
\quad \oplus \\
\quad (\lambda\, p : Policy \mid \\
\qquad (\exists\, s : p.statements \bullet EvalStatement(r)(s) = Deny) \bullet Deny)
\end{array}
$$

If a request matches the constraints of a statement, then the evaluation results in the value of the effect attribute, otherwise it evaluates to soft deny. For a request to match the constraints of a statement, the principal, action and resource of the request must each be contained in the corresponding set in the statement; in addition, all of the conditions must be met.

$$
\begin{array}{|l}
EvalStatement : Request \rightarrow Statement \rightarrow Effect \\
\hline
\forall\, r : Request \bullet EvalStatement(r) = \\
\quad (\lambda\, s : Statement \bullet SoftDeny) \\
\quad \oplus \\
\quad (\lambda\, s : Statement \mid (r, s) \in MatchStatement \bullet s.effect)
\end{array}
$$

$$
\begin{array}{|l}
MatchStatement : Request \leftrightarrow Statement \\
\hline
MatchStatement = \\
\quad \{r : Request;\ s : Statement \mid \\
\qquad r.principal \in s.principals \land \\
\qquad r.action \in s.actions \land \\
\qquad r.resource \in s.resources \land \\
\qquad (\forall\, c : s.conditions \bullet (r, c) \in MeetsCondition)\}
\end{array}
$$

To meet a condition, each of the clauses must be met using the matching relation specified by the type attribute. The key of the clause defines which of the request keys to use for each clause. For a match to be made, at least one of the clause values must match the key value.

$$
\begin{array}{|l}
MeetsCondition : Request \leftrightarrow Condition \\
\hline
MeetsCondition = \\
\quad \{r : Request;\ c : Condition \mid \\
\qquad (\forall\, cl : c.clauses \bullet \\
\qquad\quad (\exists\, v : cl.values \bullet (r.keys(cl.key), v) \in c.type.match))\}
\end{array}
$$

4 Alloy model

Having presented our Z description of the AWS access policy language, we now consider an Alloy representation, which is, via the Alloy Analyzer, more amenable to automatic analysis. We also describe domain-specific extensions for the SQS service. In Section 5 we will use the extended model both to find example policies and to test the properties of existing policies.

The domain-specific extensions are based around the types of values that can be used in a policy, the matching relations for the values, and the request keys used to access the values. In addition, the actions that can be performed are domain-specific. All of these are entities declared using abstract signatures.

```
abstract sig Value, Key, Action {}
abstract sig CondType {
  match : Value -> Value
}
```

The remaining signatures required for the policies are translated directly from the Z model.

```
sig PolicyId {}
abstract sig Version {}
one sig v1 extends Version {}
sig Policy {
  version : Version,
  pid : PolicyId,
  statements : some Statement
}
sig StatementId, Principal, Resource {}
abstract sig Effect {}
one sig Allow, Deny, SoftDeny extends Effect {}
sig Statement {
  sid : StatementId,
  effect : Effect,
  principals : some Principal,
  actions : some Action,
  resources : some Resource,
  conditions : set Condition
}
sig Condition {
  type : CondType,
  clauses : some Clause
}
sig Clause {
```

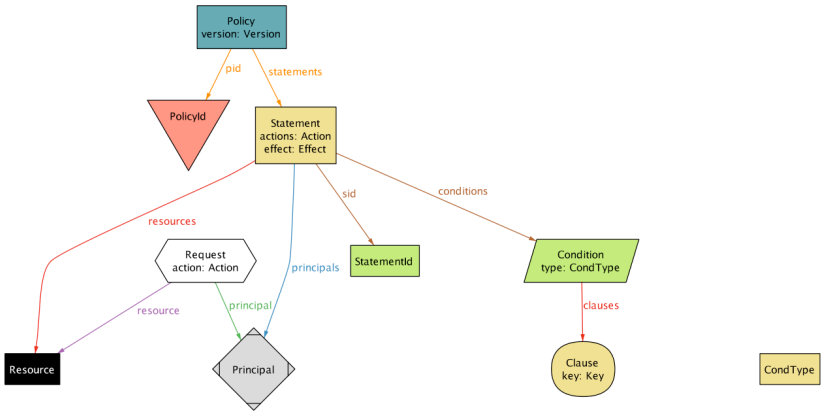


Fig. 1. Policy metamodel

```
  key : Key,
  values : some Value
}
sig Request {
  principal : Principal,
  action : Action,
  resource : Resource,
  keys : Key -> one Value
}
```

The metamodel of the basic policy signatures is shown in Figure 1.

The evaluation logic for the Alloy model is also a translation of that of the Z model.

```
fun EvalPolicies ( r : Request, ps : set Policy ) : Effect {
  (some p : one ps | EvalPolicy[r,p] = Deny) => Deny else
  ((some p : one ps | EvalPolicy[r,p] = Allow) => Allow else Deny)
}
fun EvalPolicy ( r : Request, p : Policy ) : Effect {
  (some s : one p.statements | EvalStatement[r,s] = Deny) => Deny else
  ((some s : one p.statements | EvalStatement[r,s] = Allow) => Allow else SoftDeny)
}
fun EvalStatement ( r : Request, s : Statement ) : Effect {
  MatchStatement[r,s] => s.effect else SoftDeny
}
pred MatchStatement ( r : Request, s : Statement ) {
  r.principal in s.principals
  r.action in s.actions
  r.resource in s.resources
  all c : one s.conditions | MeetsCondition[r,c]
}
pred MeetsCondition ( r : Request, c : Condition ) {
  all cl : one c.clauses |
    some v : one cl.values | (r.keys[cl.key] -> v) in c.type.match
}
```

SQS policies support five types of actions (`ReceiveMessage`, etc.—as listed in Section 2). The request keys are: `CurrentTime`, which is a date-time value;

`SecureTransport`, which is a Boolean value; `SourceIP`, which is an IP address value; and `UserAgent`, which is a string value. For strings and IP addresses, it is possible for clauses to contain either regular expressions or IP address ranges respectively. In these cases, requests contain `IPSingle` and `StringSingle` values, and clauses contain `IPRange` and `StringRange` values—both of which contain a range attribute, which is the set of values either in the IP range or that match the regular expression.

```
one sig ReceiveMessage, SendMessage, DeleteMessage,
  ChangeMessageVisibility, GetQueueAttributes extends Action {}
one sig CurrentTime, SecureTransport, SourceIP, UserAgent extends Key {}
abstract sig Boolean extends Value {}
one sig True, False extends Boolean {}
sig DateTime extends Value {}
sig IPSingle extends Value {}
sig IPRange extends Value { range : some IPSingle }
sig StringSingle extends Value {}
sig StringRange extends Value { range : some StringSingle }
```

The condition types depend on the types of values associated with the keys used in the clauses. For Boolean values there is a single condition type called `Bool`, which represents logical equivalence. For date-time values, there are condition types for comparing date-times include equality, less than and greater than. For IP addresses, there are condition types for being within or outside an IP range. Similarly, strings have condition types for matching regular expressions, as well as equality. In order to provide ordering for date-times, a standard Alloy ordering is applied to the `Value` signature.

The definitions of three of the condition types are given below.

```
one sig DateEquals extends CondType {} { match = { d1 , d2 : DateTime | d1 = d2 } }
one sig DateLessThan extends CondType {} { match = { d1 , d2 : DateTime | lt[d1,d2] } }
one sig IPAddress extends CondType {} {
  match =  { ip1 : IPSingle , ip2 : IPRange | ip1 in ip2.range}
}
```

As well as specifying the actions, condition types and values in a policy. The SQS policy documentation also adds some constraints to the policies. These include uniqueness of identifiers and limiting each policy to a single resource. These constraints are added as facts in the Alloy model. In addition, a fact has been added to ensure that the values returned by the keys function are of the correct type.

```
fact TypedKeys {
  Request.keys[CurrentTime] in DateTime
  Request.keys[SecureTransport] in Boolean
  Request.keys[SourceIP] in IPSingle
  Request.keys[UserAgent] in StringSingle
}
```

5 Examples

In this section we give two examples of use of the Alloy model. In the first, the properties of multiple policies are tested to find potential problems caused by the use of distributed access control. In the second, we look at the feasibility of using Alloy to generate candidate policies based on a set of test cases.

5.1 Two-policy test

In this example it is assumed that there are two queues: the first is used by an application to submit jobs to an EC2 instance. The results of the jobs are placed by the EC2 instance into a second queue which is then read by the application. The use of two queues in this way is one of the basic scenarios described in the SQS documentation.

There are exactly two principals used in this example: `appPrincipal` and `ec2Principal`. The resources are restricted to the two queues `requestQueue` and `reponseQueue`.

The policy for the first queue (`policy1`) contains two statements: the first allows the application to send messages providing the current date-time is less than some unspecified value; the second allows the EC2 instance to receive messages with an identical restriction on date-time values. The policy for the second queue (`policy2`) is identical to the first with the principals reversed, so the EC2 instance can send and the application can receive.

```
-- Actors
one sig appPrincipal, ec2Principal extends Principal {}
one sig requestQueue, responseQueue extends Resource {}
one sig pid1, pid2 extends PolicyId {}
one sig sid1, sid2, sid3, sid4 extends StatementId {}
-- Policy
one sig clause1 extends Clause {} { key = CurrentTime }
one sig condition1 extends Condition {} { type = DateLessThanEquals && clauses = clause1 }
one sig statement1 extends Statement {} {
  sid = sid1 && effect = Allow && principals = appPrincipal
  actions = SendMessage && resources = requestQueue && conditions = condition1
}
one sig statement2 extends Statement {} {
  sid = sid2 && effect = Allow && principals = ec2Principal
  actions = ReceiveMessage && resources = requestQueue && conditions = condition1
}
one sig policy1 extends Policy {} {
  version = v1 && pid = pid1
  statements = statement1 + statement2
}
one sig statement3 extends Statement {} {
  sid = sid3 && effect  = Allow && principals = ec2Principal
  actions = SendMessage && resources = responseQueue && conditions = condition1
}
one sig statement4 extends Statement {} {
  sid = sid4 && effect  = Allow && principals = appPrincipal
  actions = ReceiveMessage && resources = responseQueue && conditions = condition1
}
one sig policy2 extends Policy {} {
  version = v1 && pid = pid2
  statements = statement3 + statement4
}
```

To test the policies, we assume that two requests are made: the first is a request to send a message to the request queue; the second is to receive a message from the response queue. We then test a predicate that states that there exists a principal who is permitted to perform the first request but is not permitted to perform the second request. That is to say that there exists a principal who can send a message but cannot subsequently read the results.

```
one sig request1 extends Request {} { action = SendMessage && resource = requestQueue }
```

```
one sig request2 extends Request {}  { action =  ReceiveMessage && resource = responseQueue }
pred WriteButNotRead {
  request1.principal = request2.principal
  lt[request1.keys[CurrentTime],request2.keys[CurrentTime]]
  EvaluatePolicy[request1, policy1] = Allow
  EvaluatePolicy[request2, policy2] != Allow
}
```

The analyzer can find a counter-example to the predicate but requires at least 6 values, which include, `True`, `False`, an `IPSingle`, a `StringSingle` and two `DateTime` values. The `IPSingle` and `StringSingle` values are required as there must be a value for each of the four request keys. The two `DateTime` values are needed to meet the constraint that the first request happens before the second request. The unconstrained entities of the instance are represented in tree format below.

```
clause1$0
    field key
        CurrentTime$0
    field values
        DateTime$1

request1$0
    field keys
        CurrentTime$0 -> DateTime$1
        SecureTransport$0 -> False$0
        SourceIP$0 -> IPSingle$0
        UserAgent$0 -> StringSingle$0
    field principal
        appPrincipal$0

request2$0
    field keys
        CurrentTime$0 -> DateTime$0
        SecureTransport$0 -> True$0
        SourceIP$0 -> IPSingle$0
        UserAgent$0 -> StringSingle$0
    field principal
        appPrincipal$0
```

As can be seen, one failure case is that the first request is made at the same date-time as the value in the clause. As the second request must come after the first, it will fail. It should be noted that in this instance, `DateTime$1` comes before `DateTime$0` in the ordering.

5.2 Policy creation

In this example, the assumption is that a user is trying to construct a policy called `candidate` that meets three conditions: that only the application principal can read from `queue1`; that only the EC2 principal can send messages to `queue1`; and that nobody should be allowed to delete messages, change message visibility or get queue attributes from any queue.

```
assert OnlyAppCanReceive {
  all r : Request | r.action != ReceiveMessage || r.resource != queue1 ||
    (r.principal = appPrincipal <=> EvaluatePolicy[r, candidate] = Allow)
}
assert OnlyEc2CanSend {
```

```
  all r : Request | r.action != SendMessage || r.resource != queue1 ||
    (r.principal = ec2Principal  <=> EvaluatePolicy[r, candidate] = Allow)
}
assert noDeleteChangeAttr {
  no r : Request |
    r.action in DeleteMessage + ChangeMessageVisibility + GetQueueAttributes &&
    EvaluatePolicy[r, candidate] = Allow
}
```

We could use the Alloy Analyzer to find a policy that meets all of the assertions—but it is more likely to find a set of requests that avoided the constraints than a policy that met them in all cases. Instead, our approach involves creating a set of concrete examples for which the desired result was known. Two sets of requests were created: one set, which should always be successful and another set that should always fail.

```
one sig request1 extends Request {} { action =  ReceiveMessage && principal = appPrincipal }
one sig request2 extends Request {} { action = SendMessage && principal = ec2Principal  }
fun success : Request { request1 + request2 }
one sig request3 extends Request {} { action =  ReceiveMessage && principal = ec2Principal  }
one sig request4 extends Request {}  { action = SendMessage && principal = appPrincipal  }
fun failure : Request { request3 + request4 }
pred FindPolicy(generated : Policy) {
  all r : success | EvaluatePolicy[r, generated] = Allow
  all r : failure | EvaluatePolicy[r, generated] != Allow
}
```

The `FindPolicy` predicate can be used to find a candidate policy which gives the correct results for the examples given. The candidate policy can then be tested against the three original constraints to find examples of requests for which it gives incorrect results. These requests can then be added to the list of examples and a new candidate policy generated. For example, after the first iteration, the following request should have resulted in success but did not; as such, it was added to the set of successful requests.

```
one sig request5 extends Request {} {
  action = ReceiveMessage && principal = ec2Principal && resource = queue1
  keys[CurrentTime] = dateTime0 && keys[SecureTransport] = True
  keys[SourceIP] = ipSingle0 && keys[UserAgent] = stringSingle0
}
```

After six iterations, a candidate policy is found for which all three predicates hold when tested up to a size of 20.

```
one sig sid0 extends StatementId {}
one sig statement0 extends Statement {} {
  actions = ReceiveMessage
  conditions = none
  effect = Allow
  principals = appPrincipal
  resources = resource0
  sid = sid0
}
one sig sid1 extends StatementId {}
one sig statement1 extends Statement {} {
  actions = SendMessage
  conditions = none
  effect = Allow
  principals = ec2Principal
  resources = resource0
```

```
  sid = sid1
}
one sig pid0 extends PolicyId {}
one sig candidate extends Policy {} {
  pid = pid0
  version = v1
  statements = statement0 + statement1
}
```

This approach clearly has its limitations: in the general case, there is no guarantee that the number of requests required to generate a policy that meets the constraints will be of a practical size. However, if it is possible to create a policy that meets the constraints which contains a small number of statements and conditions, then the required number of tests is likely to be sufficiently small. Certainly, we envisage this test-based approach as being accessible to policy writers, and having the potential to provide some degree of formal assurance in policy construction.

6 Discussion

Cloud computing is a new computing paradigm which is gaining popularity. Computing systems built within a cloud infrastructure are constructed from multiple interacting sub-components, and access control languages can be used to restrict the interaction between these sub-components.

We have built formal models of the access policy language used within the Amazon Web Services cloud computing infrastructure. Specifically we have explored policies written for the Simple Queue Service. Using the Alloy Analyzer we have been able to explore properties of specific combinations of policies. We have also been able to use the Alloy Analyzer to assist in the construction of new policies by using sets of requests which result in known access control decisions.

As access control decisions are a security-critical function it is important that policy writers have some degree of assurance with respect to their correctness. Previous work in this area has centred around simple access control systems such as Role-Based Access Control [9, 10]. Attempts at modelling the significantly more complex XACML (see, for example, [11], [4], and [12]) have all resulted in partial models which avoid some of the more complex elements such as conditions and/or XPATH queries. In this paper we have presented a model of the whole language (which is, admittedly, less complex than XACML) that is suitable for analysis. By providing a model of the whole language it becomes possible to analyse existing real-world systems instead of placing restrictions on future systems so that analysis will be possible. This gives the model wide applicability to the rapidly increasing number of systems built using Amazon Web Services.

While the Alloy Analyzer can find instances that meet a complex set of predicates, it is not capable of simultaneously finding an instance of a policy and testing its properties against all possible requests of bounded size. Instead it will deliberately avoid requests that would break the predicates and a hence will find a policy of limited application. By separating the two steps, it is possible to build

up a set of requests that will test different aspects of the predicates resulting in increasingly applicable candidate policies. By testing candidate policies against the original predicates new requests can be found which test different aspects of the predicates.

One area of further work is to combine the model of the AWS policy language with similar models for RBAC and XACML. This will give the opportunity to explore the relationships between the models, with an ultimate goal of the automatic translation of policies from one language to another. It will also be useful when modelling the types of complex heterogeneous systems that are encouraged by cloud computing. A second avenue of further work is to consider the relationship between the Alloy and Z models (with a more concrete Z model capturing the specifics of SQS policies). In particular, we will explore the mutual benefits to be afforded through a combination of model finding and theorem proving.

References

1. Weiss, A.: Computing in the Clouds. netWorker **11**(4) (2007) 16–25
2. Amazon.com: Amazon Web Services. http://aws.amazon.com/ (2009)
3. Amazon.com: Amazon Simple Queue Service Developer Guide (API Version 2009-02-01). http://docs.amazonwebservices.com/AWSSimpleQueueService/2009-02-01/SQSDeveloperGuide (2009)
4. Zhang, N., Guelev, D.P., Ryan, M.: Synthesising verified access control systems through model checking. Journal of Computer Security **16**(1) (2007) 1–61
5. Bryans, J., Fitzgerald, J.S.: Formal Engineering of XACML Access Control Policies in VDM++. In Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M., eds.: ICFEM. Volume 4789 of Lecture Notes in Computer Science., Springer (2007) 37–56
6. Woodcock, J., Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)
7. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering Methodologies **11**(2) (2002) 256–290
8. Godik, S., Moses, T.: Extensible Access-Control Markup Language (XACML) version 1.0. Technical report, OASIS (2003)
9. Zao, J., Wee, H., Chu, J., Jackson, D.: RBAC schema verification using lightweight formal model and constraint analysis. In: Proceedings of 8th ACM symposium on Access Control Models and Technologies (SACMAT). (2003)
10. Power, D.J., Slaymaker, M.A., Simpson, A.C.: On formalizing and normalizing role-based access control systems. The Computer Journal **52**(3) (2009) 305–325
11. Bryans, J.: Reasoning about XACML policies using CSP. In: Proceedings of the 2005 Workshop on Secure Web Services. (2005) 28–35
12. Hughes, G., Bultan, T.: Automated verification of access control policies using a SAT solver. International Journal on Software Tools for Technology Transfer (STTT) **10**(6) (2008) 503–520

ParAlloy: Towards a Framework for Efficient Parallel Analysis of Alloy Models

Nicolás Rosner, Juan P. Galeotti,
Carlos G. Lopez Pombo, and Marcelo F. Frias

Department of Computer Science, FCEyN, Universidad de Buenos Aires,
e-mail: {nrosner, clpombo, jgaleotti, mfrias}@dc.uba.ar

Alloy [Jac02a] is a widely adopted relational modeling language. Its appealing syntax and the support provided by the Alloy Analyzer [Jac02b] tool make model analysis accessible to a public of non-specialists. A model and property are translated to a propositional formula, which is fed to a SAT-solver to search for counterexamples. The translation strongly depends on user-provided bounds for data domains called scopes – the larger the scopes, the more confident the user is about the correctness of the model. Due to the intrinsic complexity of the SAT-solving step, it is often the case that analyses do not scale well enough to remain feasible as scopes grow.

ParAlloy exploits the possibility of splitting the SAT formula, thus allowing for parallel SAT-solving of Alloy models. Three of its important characteristics are:

1. Its core component is a parallel solver for arbitrary propositional formulas –not necessarily in CNF– based on problem decomposition, and making a novel use of BEDs [AH02] for subproblem representation and manipulation, Minisat [ES03] for subproblem analysis, and MPI [SOHL$^+$98] for inter-process communication.
2. Its Alloy-specific enhancements further improve (parallel) analyzability by using knowledge obtained from the models to assist splitting decisions.
3. For valid properties (the UNSAT case), the speedups allowed the analysis of Alloy properties (such as some assertions in [Zav06]) that exceed the current capabilities of the Alloy Analyzer. For invalid properties, test case generation or iterative model refinement (the SAT case), parallel analysis of search space paths often leads to much higher speedups, since its exhaustion is unnecessary.

References

[AH02] Henrik Reif Andersen and Henrik Hulgaard. Boolean expression diagrams. *Information and computation*, 179(2):194–212, 2002.

[ES03] Niklas En and Niklas Sörensson. An extensible sat solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Proceedings of 6th. International Conference on Theory and Applications of Satisfiability Testing, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Santa Margherita Ligure, Italy, May 2003. Springer-Verlag.

[Jac02a] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[Jac02b] Daniel Jackson. *A micromodels of software: Lightweight modelling and analysis with Alloy.* Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 2002.

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The complete reference.* MIT Press, 1998.

[Zav06] Pamela Zave. Compositional binding in network domains. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings of Formal Methods 2006: the 14th. International FME Symposium*, volume 4085 of *Lecture Notes in Computer Science*, pages 332–347, Hamilton, Canada, August 2006. Springer-Verlag.

A Case for Using Data-flow Analysis to Optimize Incremental Scope-bounded Checking (Informal Extended Abstract)

Danhua Shao Divya Gopinath Sarfraz Khurshid Dewayne E. Perry
The University of Texas at Austin
{dshao, dgopinath, khurshid, perry}@ece.utexas.edu

Abstract. Given a program and its correctness specification, scope-bounded checking encodes control-flow and data-flow of *bounded* code segments into declarative formulas and uses constraint solvers to search for correctness violations. For non-trivial programs, the formulas are often complex and represent a heavy workload that can choke the solvers. To scale scope-bounded checking, our previous work introduced an *incremental* approach that uses the program's *control-flow* as a basis of partitioning the program and generating several sub-formulas, which represent simpler problem instances for the underlying solvers. We have developed a new approach that optimizes incremental checking using the program's *data-flow*, specifically *variable-definitions*. We expect that splitting different definitions of the same variable into sub-programs will reduce the number of variables in the resulting formulas and the workload to the backend solvers will be effectively reduced.

1 Introduction

In software verification, *scope-bounded* checking [2] of programs has become an effective technique for finding subtle bugs. Given bounds (that are iteratively relaxed) on input size and length of execution paths, the code of a program is translated into a relational logic formula, and a conjunction of this formula with the negation of the post condition specification $Pre \wedge translate(Proc) \wedge \neg Post$ is solved using off-the-shelf SAT solvers. A solution to this formula corresponds to a counterexample.

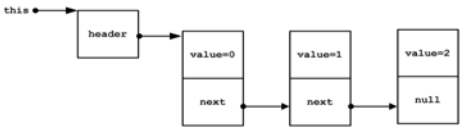
Traditional scope-bounded checking [1] translates the bounded code segment of the *whole* program into *one* input formula. For non-trivial programs, the translated formulas can be quite complex and the solvers can fail to find a counterexample in a desired amount of time. When a solver times out, typically there is no information about the likely correctness of the program or the coverage of the analysis completed.

Recently, we introduced an *incremental* approach based on the program's *control-flow* to increase the efficiency and effectiveness of scope-bounded checking [3]. The key idea is to partition the set of executions of the bounded code fragment into a number of subsets and encode each subset into a sub-formula. We *split* the program into smaller sub-programs, which are checked according to the correctness specification. Thus, the problem of scope-bounded checking for the given program reduces to several sub-problems, where each sub-problem requires the constraint solver to check a less complex formula.

The splitting strategy in our previous work [3] focuses solely on the program's *control-flow,* and is therefore limited to the syntactical structure of the program and fails to exploit the program semantics.

```
public class IntList {
    private Entry header = null;
    private static class Entry {
        int value;
        Entry next;

        Entry (int value, Entry next){
            this.value = value;
            this.next = next;
        }
    };
    public boolean contains (int key) {
        Entry e = this.header;
        while ( e != null){
            if (e.value == key){
                return true;
            }
            e = e.next;
        }
        return false;
    }
}
```



(a) (b)

Figure 1. Class `IntList` (`contains`() method and an instance).

Since the complexity of the formulas comes from both the data-flow and the control-flow, we hypothesize that the use of data-flow in defining splitting strategies is likely to further reduce the workload of the constraint solvers. We introduce a splitting strategy based on *variable-definition*s. Specifically, we split the program based on different definitions of the same variable into sub-programs, which leads to a reduction in the number of variables in the resulting sub-formulas. The rationale behind this is that decrease in the number of definitions for a variable would reduce the number of intermediate variable names and thus the number of frame conditions introduced in data flow encoding.

2 Example

Suppose we want to check the `contains()` method of class `IntList` (Figure 1 (a)).

An object of `IntList` represents a singly-linked list. The `header` field points to the first node in the list. Objects of the inner class `Entry` represent list nodes. The `value` field represents the (primitive) integer data in a node. The *next* field points to the next node in the list. Figure 1 (b) shows an instance of `IntList`.

Consider checking the method `contains()` of class `IntList`. Assume a bound on execution length is one loop unrolling. Figure 2(a) shows the program and its *computation graph* [2] for this bound.

Our program splitting strategy is *variable-definition* based. Given a variable in the computation graph, we split the graph into multiple sub-graphs such that each sub-graph has at most one definition for the variable ,that can reach the exit statement.The definition of this variable in each sub-graph is different.

In Figure 2 (a), the definition of variable this and key is empty set {}. Definitions of variable return is statement set {4, 8, 11}, and definition of variable e is statement set {1, 5, 9}. All of these definitions can reach the exit statement.

Suppose we select definitions of variable e (the most modified variable) to split the computation graph, we construct three sub-programs: Figure 2(b), 2(c), and 2(d). Each sub-program only contains one definition of variable e.

3 Summary

Scalability is a key challenge for scope-bounded checking. For non-trivial programs, the formulas translated from control-flow and data-flow can be quite complex and the

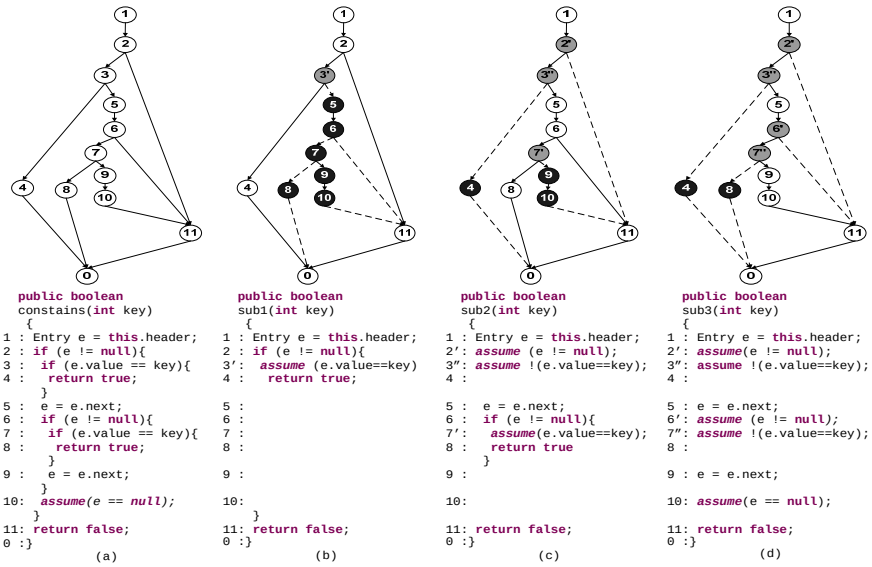


Figure 2. Splitting of program `contains`() based on definitions of variable `e`. *Broken lines* in sub-graph indicate edges removed constructing this sub-program during splitting. *Gray nodes* in a sub-graph denote that a branch statement in original program has been transformed into an assume statement. In programs below computation graph, the corresponding statements are show in Italic. *Black nodes* denote the statements removed during splitting. Subgraph (a) is program `contains`() and its computation graph after one-round unrolling. At exit, there are three definitions of variable `e`: Statement 1, 5, 9. Subgraph (b) is based on definition at statement 1. Subgraph (c) is based on definition at statement 5. Subgraph (d) is based on definition at statement 9.

heavy workload can choke the solvers. Our previous work used control-flow as a basis of an incremental approach to scope-bounded checking by splitting the program into smaller sub-programs and checking each sub-program separately, and demonstrated significant speed-ups over the traditional approach. We recently developed a new splitting strategy based on data-flow, specifically variable definitions, to optimize the incremental approach. We believe that use of variable definitions can effectively reduces the number of variables the complexity of the ensuing formulas and provides more efficient analysis.

Acknowledgment: This material is based upon work funded in part by NSF (Grants IIS-0438967, CCF-0702680, and CCF-0845628) and AFOSR (FA9550-09-1-0351).

References

[1] G. Dennis, F. S. H. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA* 2006.
[2] D. Jackson, and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA* 2000.
[3] D. Shao, S. Khurshid, and D. E. Perry. An incremental approach to scope-bounded checking using a lightweight formal method. In *FM* 2009

A Basis for Feature-oriented Modelling in Event-B

Jennifer Sorge, Michael Poppleton, Michael Butler

Electronics and Computer Science, University of Southampton
{jhs06r,mrp,mjb}@ecs.soton.ac.uk

Abstract. Feature-oriented modelling is a well-known approach for Software Product Line (SPL) development. It is a widely used method when developing groups of related software. With an SPL approach, the development of a software product is quicker, less expensive and of higher quality than a one-off development since much effort is re-used. However, this approach is not common in formal methods development, which is generally high cost and time consuming, yet crucial in the development of critical systems. We present a method to integrate feature-oriented development with the formal specification language Event-B. Our approach allows the user to map a feature from the feature model to an Event-B component, which contains a formal specification of that feature. We also present some patterns, which assist the user in the modelling of Event-B components. We describe a composition process which consists of the user selecting an instance in the feature model and then constructing this instance in Event-B. While composing, the user may also discharge new composition proof obligations in order to ensure the model is consistent. The model is then constructed using a number of composition rules.

1 Introduction

Current critical systems have become more complex and more common, which requires them to be developed more efficiently and preferably with the application of formal methods to ensure a safer system. The development with formal methods is very time-consuming and costly, so in many cases formal methods are not used. In our work we use the formal method Event-B [1], which is based on first-order logic and set theory. It is structured into a dynamic part (describing system behaviour) and a static part (describing contant data and types). The dynamic part is referred to as a machine, and the static part is called a context. Event-B is supported by the open tool Rodin [1]. In non-critical systems, it is common to use SPL development techniques to save time and develop better software faster. One such approach is feature modelling [2], which is used to structure a set of related software products into common and variable requirements, referred to as features. The feature diagram can be used to generate

[1] The continued development of the Rodin toolset is funded by the EU research project ICT 214158 DEPLOY (www.deploy-project.eu).

instances of the software family; this is done by the selection of different features within the diagram.

Currently, the development of Event-B models is a time-consuming task, which requires a lot of proof. Our motivation is to reduce the development time and to considerably reduce reproof by reusing Event-B components for which proof obligations have been discharged. During the process of composition, composition proof obligations can be discharged. By experimentation we have shown that a lot of the original proof obligations do not have to be reproved during composition, thus saving a lot of prover resources.

An extended version of this paper is available from [3].

2 Process

In Figure 1 we present the composition process. The feature model is formed by features which may be associated with Event-B components. The composition process entails a subset of features to be selected from the feature model to form a feature model instance, thereby selecting several of these Event-B components. These components are composed pair-wise, and composition proof obligations can be discharged to prove properties and to ensure consistency of the composition. The final Event-B machine represents the formal specification which is associated with the feature model instance and is obtained by composing these components.

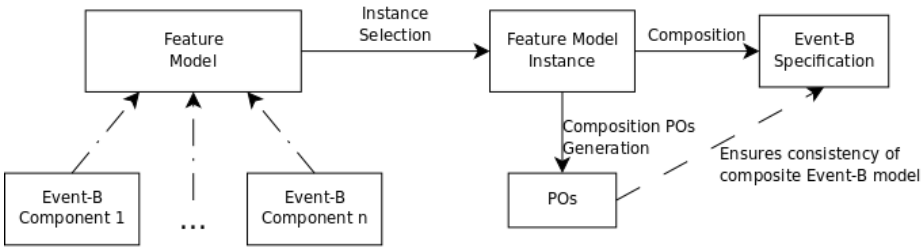


Fig. 1. Overview of Composition Process

2.1 Integration of Feature Models and Event-B

In order to link feature models with Event-B, we develop Event-B components which represent features in a feature model and that can be linked to them by name. Only leaf node features can be mapped to an Event-B component. We have developed a number of Event-B modelling patterns which provide guidelines that help the construction of single Event-B components. An Event-B component may consist of zero or more contexts, but must be consistent and independent of any other components. Refinement is also currently not supported. All proof obligations for a component must be discharged.

2.2 Proof and Composition

Composition of components is n-wise, if more than two features that are mapped to components are selected in the feature model. The composition process, however is pair-wise. This means that always two components are composed. All proof obligations for a component have been discharged during the creation of a component, however, when composing two components, new proof obligations may come up. We will refer to these as composition proof obligations. Proof obligations that are only concerned with one component are referred to as component proof obligations.

Composition proof obligations are proof obligations that can be discharged during composition of two components. They are based on component proof obligations and their task is to reduce reproof of component proof obligations.

Once composition proof obligations have been discharged, the two components can be composed. We have developed a number of composition rules that can be applied to the composition of Event-B contexts and machines. The outcome of composition is one single Event-B model.

3 Conclusion and Future Work

Our work demonstrates the integration of feature models and Event-B, thus enabling SPL development for formal methods and providing a way to prove certain properties about a composition. Currently our work is fundamentally theoretical, however we have been collaborating in the development of a Rodin plugin to integrate this theoretical approach with the Rodin platform. This plugin contains a composition tool, and in future will support feature model editing and an instance generator[4, 5].

References

1. Abrial, J.R.: Modeling in Event-B: Systems and Software Engineering. To be published by Cambridge University Press (2009)
2. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
3. Sorge, J., Poppleton, M., Butler, M.: A Basis for Feature-oriented Modelling in Event-B. (2009) http://eprints.soton.ac.uk/69645/.
4. Poppleton, M., Fischer, B., Franklin, C., Gondal, A., Snook, C., Sorge, J.: Towards Reuse with"Feature-Oriented Event-B". (2008)
5. Gondal, A., Poppleton, M., Snook, C.: Feature composition-towards product lines of event-B models. (2009)

Formal analysis in model management: exploiting the power of CZT

James R. Williams, Fiona A.C. Polack and Richard F. Paige*

Department of Computer Science, University of York, UK YO10 5DD
[jw,fiona,paige]@cs.york.ac.uk

1 Background

Software engineering diagrams can be hard to formally analyse due to inadequately defined diagram semantics; often the semantics are underspecified to a degree that does not allow useful properties to be checked. Whilst the concept of diagram verification through formalisation has a long history, there has been little success in linking approaches to software engineering process, and tool support tends to expose diagrammers to substantial formalism.

The AUtoZ tools provide formalisation in the style of commercially-acceptable model management [9,8]. AUtoZ is an automated framework based on Amálio's GeFoRME, the generative framework for rigorous model-driven engineering [1]. GeFoRME is designed to give semantically-adaptable support to the construction of formal models from diagrams. The semantics of diagram concepts (classes, associations, states etc) and the semantics of the domain (what the diagram describes) are combined through template instantiation, to derive a formal model that can be analysed using existing tools. Amálio devised the *Formal Template Language* (FTL) as the rigorous underpinning to GeFoRME, supporting proof and metaproof through template representation. A GeFoRME framework comprises a catalogue of FTL templates. Selected templates are instantiated using names of concept instances from diagrams, to generate formal models. Amálio has shown that, if the original diagrams are consistent, then the generated Z is type-correct. The templates include conjectures and proofs, and FTL underpins correctness-by-construction, enabling reasoning at the level of the templates [2].

The selection and instantiation of templates, and the presentation of generated formal specifications to analysis tools can be largely automated, along with support for template management [9,8].

2 Automatic formalisation of UML to Z: AUtoZ

AUtoZ (`www.jamesrobertwilliams.co.uk/autoz.php`) is an automated framework supporting GeFoRME UML+Z transformation. It provides formal consistency analysis, using generated conjectures and proofs, for UML class and

* This research was supported by the EPSRC, through the Large-Scale Complex IT Systems project, EP/F001096/1

state diagrams. This means that UML diagrams can be cross-checked, existence checked, and consistency checked, providing the designer with confidence in the correctness and consistency of the UML model.

Methods integration is an instance of model transformation [3,6]. This allows the design of AUtoZ to exploit best practice in model-driven engineering and model engineering tools (Eclipse UML2 and Epsilon – `www.eclipse.org/gmt/epsilon/`), and results in a formalisation that is presented to the software engineer as an optional extension to existing tool-supported model-management activities.

The AUtoZ framework supports tool specialisations built, for example, for different graphical modelling or formal analysis tools. As well as formalisation and analysis, the framework supports expert use, to extend and modify template libraries. Two instances of the AUtoZ framework are AUtoCADiZ and AUtoZ/Eves [9,8].

AUtoZ and call-outs to existing modelling and formal tools are Eclipse plug-ins. Amálio's FTL templates are translated into a library of Epsilon Generation Language (EGL) instances [5]. EGL then operates on the serialised output of a diagramming tool (any meta-model-based diagramming tool that produces serialised output can be used) to extract concepts and concept names to instantiate the templates. EGL outputs model details and necessary textual annotations to populate the templates and construct the formal model [9,8].

3 Customising communication: AUtoCZT

AUtoCADiZ and AUtoZ/Eves generate specifications in LaTeX Z markup. Whilst CADiZ and Z/Eves are powerful analysis tools, the messages that they produce are aimed at expert users of Z and the Z tool, not general software engineers. For instance, messages from the typechecker and theorem provers refer to line numbers and formal concepts in the Z specification, rather than elements of the UML models. This is a common issue in tool support for integrated methods [4].

To address the customisation of tool-generated messages, we are creating an AUtoZ instance that targets the CZT tool suite. The *Community Z Tools* (CZT) project (`czt.sourceforge.net`) is an open-source project providing tool support for Z. The *ZML* sub-project of CZT [7] introduces XML markup for Z, which can be used either as direct input to the CZT tools or to generate LaTeX markup. CZT tools annotate the ZML file, for instance with issues raised by formal analysis tools.

To customise error messages, AUtoCZT exploits the fact that, in model engineering, a diagrammatic model must conform to a meta-model (defining abstract syntax and some semantics). Models that conform to the same meta-model are comparable, and, where models conform to different meta-models, generic association can be made at the meta-model level. Epsilon provides the ModeLink tool to support such modelling activities (`www.eclipse.org/gmt/epsilon/doc/modelink`). Since UML and ZML have metamodels, a generic association can be constructed, such that elements in the UML and the Z models can be matched

using ModeLink. Traceability links are now a side-effect of the Z generation. CZT formal analysis annotations (in ZML) use the links to map to the UML model.

Figure 1 demonstrates the tool chain of AUtoCZT.

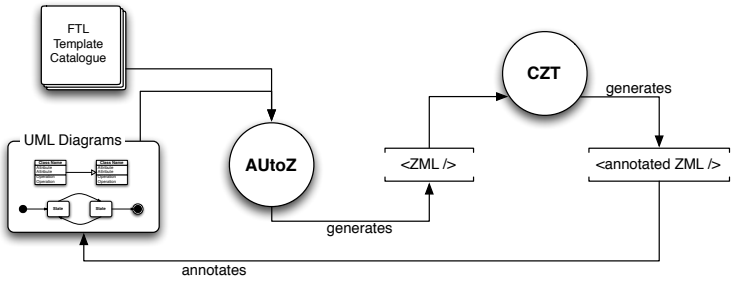


Fig. 1. The tool chain of AUtoCZT

4 Discussion

GeFoRME and AUtoZ provide frameworks for practical formal analysis of diagrammatic models. The genericity inherent in the approaches is applicable to other transformation approaches. State-of-the-art model management tools provide the basis for powerful automation. By combining the automation with CZT's flexible, open-source formal support mechanisms, a complete tool chain has been designed which can overcome many of the problems of interfacing formal analysis with traditional diagram-based software engineering.

References

1. N. Amálio. *Generative frameworks for rigorous model-driven development.* PhD thesis, Computer Science, York, UK, 2007.
2. N. Amalio, S. Stepney, and F. Polack. A formal template language enabling metaproof. In *FM 2006*, volume 4085 of *LNCS*, pages 252–267, 2006.
3. D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The Epsilon Transformation Language. In *ICMT'08*, volume 5063 of *LNCS*, pages 46–60, 2008.
4. R. Laleau and F. Polack. Coming and going from UML to B : a proposal to support traceability in rigorous IS development. In *ZB 2002*, volume 2272 of *LNCS*, pages 517 – 534, 2002.
5. L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack. The Epsilon Generation Language. In *EC-MDA'08*, volume 5095 of *LNCS*, pages 1–16, 2008.
6. H. Treharne, E. Turner, R. F. Paige, and D. S. Kolovos. Automatic generation of integrated formal models corresponding to UML system models. In *TOOLS Europe '09*, volume 33 of *LNBIP*, pages 357–367, 2009.
7. M. Utting et al. ZML: XML Support for Standard Z. In *ZB 2003*, volume 2651 of *LNCS*, pages 437–456, 2003.
8. J. Williams and F. Polack. Automated formalisation for verification of diagrammatic models. In *FACS'09*, ENTCS, 2009.
9. J. R. Williams. AUtoZ: Automatic formalisation of UML to Z. MEng thesis, Computer Science, York, UK, 2009. `www.jamesrobertwilliams.co.uk/publications/WilliamsMEng.pdf`.

Starting B Specifications from Use Cases

Thiago C. de Sousa[1] and Aryldo G. Russo Jr[2]

[1] University of São Paulo
thiago@ime.usp.br
[2] AeS Group
agrj@aes.com.br

Abstract. The B method is one of the most used formal methods, when reactive systems is under question, due to good support for refinement. However, obtaining the formal model from requirements is still an open issue, difficult to be tackle in any notation due the background distance between the requirement engineer and the one in charge of work with a formal specification. On the other hand, use cases have become the informal industry standard for capturing how the end user interacts with the software by detailing scenario-driven threads. Furthermore, the scenarios steps provide an easy way to derive functional tests, as in the same way what has to be achieved and what's not is, normally, clearly stated. In this paper we show how controlled use cases and functional tests based on them can be used as a guideline for writing B operations and invariants. As a side effect, we also present a practical way to establish traceability between functional requirements and formal models.
Keywords. B method, requirements traceability, use cases transactions

1 Introduction

B [1] is a formal method that allows us to produce proof obligations that demonstrate correctness of the design and the refinement. Nevertheless, there is no standard mechanism for mapping requirements to formal specifications. To overcome this issue, different solutions have been proposed by researchers. In [2], the authors have presented a traceability between KAOS requirements and B. Some authors are investigating the use of the Problem Frames approach [3] as a possible response. A mixed solution using natural language and UML-B has been proposed by [4]. However, these approaches use non-standard artifacts for requirement specifications, which we consider a disencentive for convincing designers to adopt formal methods since they must spend time to learn them.

Use cases [5] can be considered as the *de-facto* industry standard for requirement specifications. They provide a good way to capture how the end user interacts with the system by detailing scenario-driven threads. A typical use case describes a user-valued transaction in a sequence of steps expressed in a natural language, which makes use cases readable for most end-users. In [6] the author has presented an approach for building B specifications from use-case models. This approach is similar to ours, but his project has focused on bringing

the object-oriented paradigm (including UML diagrams) to formal methods. His method also maps each use case as a unique B operation, what we believe is not correct since each use case can have many transactions according to Jacobson[7].

Another relevant point about uses cases is the possibility to derive test cases. One of the new proeminent development model is the so-called Test Driven Development[8], where the input information used to generate the source code are the test cases, instead of use case scenarios or other traditional requirement documentation.

In this paper we propose an approach for starting B specifications from use and test cases. Use cases transaction identification can be used as a guideline for defining B operations, including the pre- and postconditions. In the same way, test cases can help on the definition of global invariants and constraints. This research is part of a bigger project, called *BeVelopment*[9], partially supported by DEPLOY project. This project is presented in the section 2.

In the section 3 we explain our approach in more details showing how it works on an example for booking flights for traditional B and a small (industrial) example used to explain our approach for Event B. In the section 4 we introduce a possible utilization of annotation techniques to facilitate the process. This annotation process can be viewed as a refinement in a informal way, were constraints and supported information are added to contribute for a better understanding of the problem, and to help in the formal refinement steps.

Finally, we reserve the last section for further discussions and enumerate some of the future works.

2 BeVelopment Project

As can be seen in figure 1, the development process is composed by several phases and each phase composed by several tasks. This model was extracted from the IEC 61508[10] standard, although it's similar in several different fields of application.

Most of the time, when this model is followed, in safety critical applications, formal methods are used only in a small part of the process. Our objective here is to spread formal method utilization in almost all the phases in this process, and ultimately, create a guide of application that could be used by others in order to introduce this extended methodology.

For that, we intend to use what's been developed inside DEPLOY workpackages (see figure 1 for details) to create a chain of application.

We expect to be able to integrate what's been developed in Workpackage 1 and 4 during the system and software specification phase. The effort here will be in determine a way to specify better requirements to avoid that errors like ambiguities and inconsistencies move forward to later phases. There are also other studies we'd like to investigate in this phase like in [11] and [12]

During the software architecture and design we intend to use what's been developed in workpackage 2 and 3 along with the decomposition technique that's on development, trying with that, in the determined point separate hardware

part and software part, and from that continue with the next refinements independently from each other.

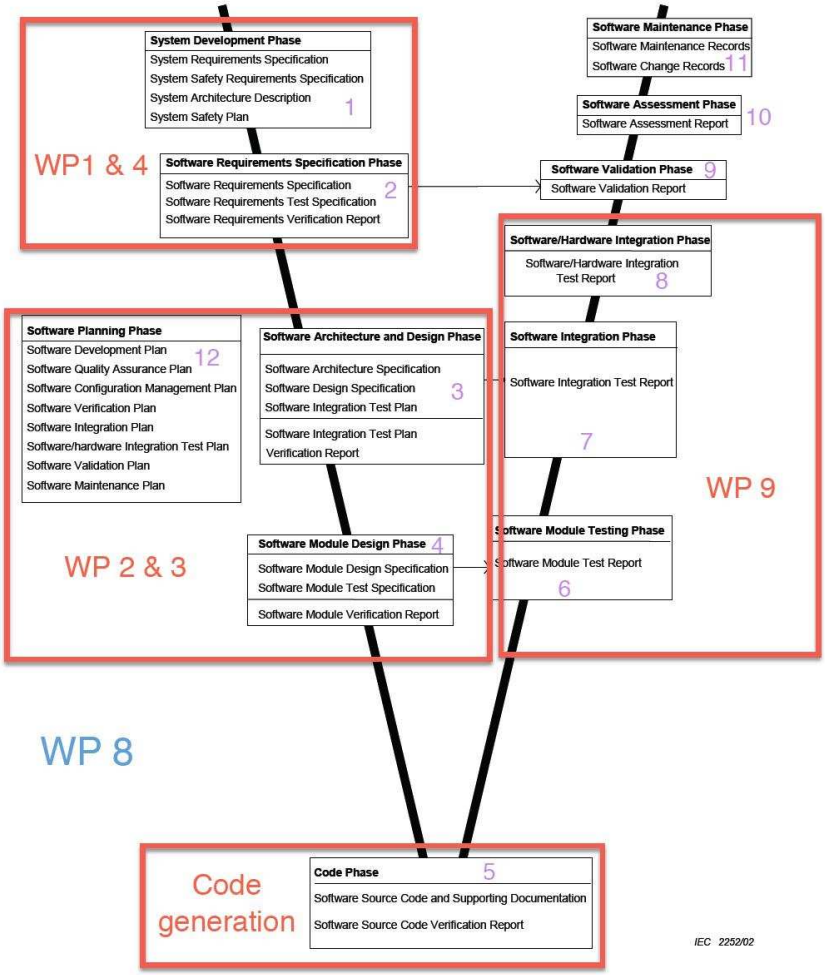


Fig. 1. V model and Workpackages

After code generation phase, that we hope it would be possible to be done automatically from the refined specification, we intend to use what's been developed in workpackage 9, in the sense of helping the tool development group in create an appropriate plug-in to extract test cases from the formal specification.

At the end, the whole process is part of a more wide objective that is improve the dependability of the developed product, which meets the expectation of the workpackage 8.

2.1 Project details

The main objective of this project is to create a useful methodology to be used during the development life cycle of safety critical systems. In order to do that, is a fact that some ingredients are needed, as follow:

1. A development life cycle *framework*. This framework must define what are the phases in this life cycle, and what are the inputs and outputs of these phases. Moreover, it needs to state what are the tasks that need to be performed to "transform" the inputs in the correspondent outputs of each phase. There are several frameworks that could be used, like spiral, clean room, XP, etc.. but, as it is the case of railway domain, the V model would be the one used in this project.
2. For the "transformation", or to perform each task, it's necessary some *techniques* (or languages, tools, etc...) that would be used to get the inputs and generate the expected outputs. As the objective of this project is the application of formal methods (in our understating, formal methods are, in fact, formal languages by the fact of lack of a utilization method, like is stated in [13]) during the development life cycle, one of the expected results is the identification of what method and related tools would be suitable. As in some other previous studies, we have strong feelings that the B method and its derivatives would suit well in most of the cases in railway domain. But, where is the case of necessity other formalisms would be applied, and as a secondary objective in this project we would like to evaluate, compare, and verify other methods like VDM [14], Z [15], and others.
3. But, to be able to use such techniques, an utilization (or application) method need to be used in order to guide, or to state the steps that are necessary to successfully achieve the objectives. In almost all cases, such formal languages are not followed with this methods, and, as was stated by Jens Bendisposto and Michael Leuschel, during Dagstuhl Seminar (Refinement Based Methods for the Construction of Dependable Systems), using the example of the "Abrial index", where can be seen that when these formal languages are used by people that really knows about that quite well, the resulting specification is easily proved where is not the case when it's done by people who not follow a (hidden) method. During this project, as another secondary objective, we would like to determine a method that would guide these techniques application, to help people during the development life cycle to spend their time in valuable tasks, and not in "try and error" experiences.
4. Finally, as the *methodology* itself, is the task to determine the transitions from one phase to another. It needs to be a guide that state what intermediate tasks are needed in order to an output of a previous phase could be used as input of the next one. Moreover, it's the translation of the used framework

in words that state how to perform whatever is needed to achieve the end of the life cycle, and not only the "what" needs to be done. This is the main objective of this project, and we hope it could be generic enough that could be used in other domains, but strong enough to facilitate the adoption of formal methods in railway domain as a strong methodology that helps the accomplishment of what is already required by the domain standards.

The work presented in this paper is related to the system analysis and physical model, presented in 1. At this stage it's not our aim an automatic translation (although it's likely to be a necessity in the near future) from the use cases to the formal model, instead, we intend to facilitate the manual process.

3 Mapping UC to B

The aim of this approach is to fulfill some intermediate phases in the proposed development process. Until now, it's commonly seen the use of formal methods in the development process, only from the design phase where the requirements are already well defined, but this is not the case on industrial projects.

Based on that, the method proposed here would be useful to be used during early phases, to help on both directions:

- to the top, helping the requirements elicitation (what might be combined with other techniques and methods, like Jackson's Problem Frames)
- to the bottom, helping the creation of the first abstract formal model, in the sense that it can support the first definitions.

For mapping use cases to traditional B specifications we propose that use case scenario sentences must be written using a controlled natural language (CNL) described according our use case transaction definition, which is based on Ochodek's transaction model [16].

Definition 1. *A transaction is a shortest sequence of actor's and system actions, which starts from the actors request and finishes with the system response. The system validation and system expletive actions must also occur within the starting and ending action. The pattern for a transaction written as a sequence of four steps in a scenario:*

n. An actors request action (U).
n+1. A system data validation action (SV).
n+2. A system expletive action (e.g. system state change action) (SE).
n+3. A system response action (SR).

We have also decided to define the grammar using subject-verb-object (SVO) sentences because they are good at telling the sequence of events. We have mapped the use case actor as subject, a set of actions predicate synonyms (for example validate, verify etc. would be grouped together) as verb and the rest of the sentence as object.

For the sake of simplicity, we assume that all environmental tasks, like user interactions, are perfect, i.e., they are executed always in a correct order and in a correct moment. Further studies will be developed to incorporate non-functional requirements and exceptions.

Let's take a look at an example in order to clarify our idea, first about the generation of controlled use cases, then we introduce the annotation and finally, the test cases generation. In accordance with previous definitions, suppose we have the following use case scenario:

```
1. The agent specifies a travel itinerary for a client.
2. The system validates the itinerary.
3. The system searches a set of appropriate flights.
4. The system presents them to the agent.
5. The agent selects a flight.
6. The system verifies free spaces on the flight.
7. The system reserves any seat from the set of free spaces.
8. The system confirms the reservation.
```

In the above example, we can see two transactions (1-4 and 5-8) and each one can be used as a guide to create a B operation. From SV actions (2 and 6) we extract the preconditions (validates the itinerary, verifies free spaces) and from SE actions (3 and 7) we derive the operations names (search, reserve) and the postconditions (searches a set of flights, reserves any seat). A possible mapping to B specifications, which is self-explanatory, is shown below.

flights ← search(city1,city2) ≙
  PRE city1 ∈ CITY ∧ city2 ∈ CITY ∧ city2 ∈ itinerary(city1) **(2)**
  THEN flights := companies({city1 ↦ city2}) **(3)**
  END;

reserve(flight) ≙
  PRE card(flight) > 0 **(6)**
  THEN ANY seat **WHERE** seat ∈ flight **(7)**
    THEN flight := flight - {seat}
    END
  END

For mapping use cases to Event B specifications we must include one more step in the previous definition.

Definition 2. *A transaction is a shortest sequence of actor's and system actions, which starts from the actors request and finishes with the system response. The system guard recognition, validation and expletive actions must also occur within the starting and ending action. The pattern for a transaction written as a sequence of four steps in a scenario:*

n. An actors request action (U).
n+1. A system guard recognition action (SG)
n+2. A system data validation action (SV).
n+3. A system expletive action (e.g. system state change action) (SE).
n+4. A system response action (SR).

Another example, to explore the abstraction of the first model for Event B, can be seen bellow. This is an example based on a train door system, where, when a open command is received by the system, if the conditions are satisfied, the train door must open. This is an attempt to use this approach at an abstraction level, but it seems to be strong enough to present the general information of the system.[3]

```
1. The agent requests to open the doors on one side of the train.
2. The system recognise the request.
3. The system verify the validity of this request.
4. The system commands the opening.
5. The system confirms the execution.
```

We can use SG, SV and SE actions as a guide to derive guards, conditions and actions. A possible mapping for an Event B model was created from this use case:

OPEN_COMMAND $\triangleq$
WHEN grd1: REQUEST = TRUE **(2)**
WITH cond: CONDITIONS = TRUE **(3)**
THEN act1: OPEN := TRUE **(4)**
END
END

At this point we just present superficial information, complete enough to understand the global functionality of the system. Again, at this point there is no need to present:

- how the system recognize the request;
- how it must to be validate;
- how the command is generated.

These suppressed information help to understand the main goal of the system, and there is no need to express the internal behavior of the controller itself. In the next section we present a possible approach to introduce such information.

4 Informal UC Refinement

The second step(under development), in order to generate a full B specification, is the generation of test cases, based on use cases specification.

[3] as this is highly an abstract representation of the (one function of) the system, a lot of manual work has to be performed, like variable names, etc...

As stated before, we also intend to use test cases to help on the development of formal specification, but before that, some test cases classification is needed to help on understanding the proposition.

One possible classification of tests are:

- *test to succeed* - these are tests that can validate that the system is performing it's actions accordly with what was specified for the system to do, meaning that, based on the correct inputs, the system must provide the correct output.
- *test to fail* - these are tests that are used to verify the behavior of the system when inputs are provided in a different sequence, order, or value. It's used to check if in those cases the system has the correct protection to avoid misunderstandings or dagerous behaviors.

Based on this classification, it's not difficult to see that the generation of the first one is almost a straightforward categorization of the use cases, but the second one is not trivial.

To help the creation of the second family of test cases, we propose an annotation technique that must be applied to the use cases in order to facilitate the definition of bounderies, types, and any other information that could determine possible exceptions.

as stated before, from one single use case, it's possible to derive several test cases, divided in test to succeed, and test to fail. to help this generation, we propose a simple, informal annotation system that needs to be included in the use case specification, do drive the user to determine what test cases are necessary. Basically, this annotation system cover the following properties:

- Boundaries
- Possible values
- Safety properties (like, not allowed sequences, wrong events, etc...)

An example for this annotation (based on the first example) can be seen bellow:

```
  1. The agent specifies a travel itinerary for a client.
 [AN1:-> the source and destination must exists.
  AN2:-> the valid characters are only alphabetical]
  2. The system validates the itinerary.
```

The generation of the test case related to test to succeed are straightforward, so it will not be cover here, but to generate the test cases related to test to fail, the included information (the developer/client wishes), can help a lot.

One example of test to fail based on this annotaded use case can be:

```
  1. The agent specifies, as a travel itinerary, the follow data:
  Paris12.
  2. The system verify the validity of this itinerary.
  3. The system return the related error message.
 (as the property, in the annotation part, do not hold)
```

More formally, the refinement process is based on the horizontal refinement techniques [17], where new information is introduced in each step. Of course, these refinements can not be proved as correct in the Use Cases phase, but it will be verified during the formal development. Moreover, it's a complimentary tasks that needs to be reviewed in case of failures during the verification.

Based on the second example, we can have the first refinement, based on annotations, like the one presented bellow:

```
 1. The agent requests to open the doors on one side of the train.
[AN1 - this operation is made by selection
the side and pressing the correspondent bottom]
 2. The system recognise the request.
[AN2 - this operation is only allowed if the train
is stopped in the correct position(at the station)]
 3. The system verify the validity of this request.
[AN3 - all the conditions must be true at the same time]
 4. The system command the opening.
[AN4 - at least 2 different signals must be
activated to guarantee the safety condition]
```

From this example, we can see that's possible derive several test cases. Some examples are presented bellow. Again, first we present the test to succeed, and after some possibilities of test to fail. One test to succeed would be:

```
1. The agent select the right side for opening
 and press the right open bottom.
2. The system verify the sequence and the consistency,
 i.e., if the side and bottom are related,
 and if the speed is 0km/k
3. The system activate both output signals
```

This procedures can be done in two different flavors. In one side we can work on both, informal (use and test cases) and formal model at the same time, I mean, at the moment that one refinement is made in the informal model, it must be reflected in the formal one. This approach helps the mistakes discovery during the early development phases, although it brakes the reasoning flow during the construction of informal model. On the other hand, we can refine completely the informal model, and after generate the formal one. As it can help the reasoning flow, it can be a problem if errors are discovered later in the process. Based on these information, a weighting must be performed to determine when and how go to formal model and come back.

The last step, the generation of global invariants and constraints is still under study and it's supposed to be presented in a future revision of this work.

5 Discussions and Future Work

In this paper we have proposed an approach for mapping requirements to B (Event B) models through use and test cases in a pragmatic way. We are not

interested (at this moment) in the automatic translation of use cases for formal specifications since there are many natural language ambiguity problems. The intention is to take the use cases as a guideline for starting B specifications.

Our main goal is to create a new and complete development process (including deliverables artifacts), namely *BeVelopment*, for B focusing on agility/usability and we believe that use cases seem to be a good start point.

Other approaches are under study right now like Problem frames and KAOS. The intention is to facilitate as much as we can the beginning of the development process where formal methods are supposed to be used.

The project is in the initial phase and there are a lot of future works. First, we are planning to use the Rodin platform as tool support. Another possible improvement would be a more flexible grammar since there are B operations without preconditions. Alternative scenarios (extensions) must also be included in the transaction discovery process. At the moment, our approach only maps B operations (events for Event B) and we are investigating other artifacts for extracting the B machine name, constants, properties, assertions, variables, initializations and invariants as well as non-functional requirements, including performance and reliability.

6 Acknowledgments

We'd like to thank to Prof. Ana C. V. Melo, who devoted his time to review this paper and mostly for her comments which help us to include substantial points in this work.

References

1. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York, NY, USA (1996)
2. Ponsard, C., Dieul, E.: From requirements models to formal specifications in B. ReMo2V CEUR Workshop Proceedings (2006)
3. Jackson, M.: Problem Frames: analyzing and structuring software development problems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
4. Jastram, M., Leuschel, M., Bendisposto, J., Jr, A.R.: Mapping requirements to B models. DEPLOY Deliverable - Unpublished manuscript (2009)
5. Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley Professional (1992)
6. Ledang, H.: Automatic translation from UML specifications to B. Automated Software Engineering Conference (2001)
7. Jacobson, I.: Object-oriented development in an industrial environment. In: OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications, New York, NY, USA, ACM (1987)
8. Beck, K.: Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
9. Jr, A.R.: DA associate program - AeS proposal. Unpublished manuscript (2009)

10. Bell, R.: Introduction to IEC 61508. In: SCS '05: Proceedings of the 10th Australian workshop on Safety critical systems and software, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2006) 3–12
11. Lamsweerde, A.v.: Goal-oriented requirements engineering: a guided tour. Proceedings of Fifth IEEE International Requirements Engineering Symposium (2001) 249–262
12. Gunter, C.A., Gunter, E.L., Jackson, M., Zave, P.: A reference model for requirements and specifications. IEEE SOFTWARE **17**(3) (2000) 37–43
13. Mazzara, M.: Deriving specifications of dependable systems: toward a method. In: Proceedings of the 12th European Workshop on Dependable Computing (EWDC). (2009)
14. Bjørner, D., Jones, C.: The Vienna development method: The meta-language. Springer-Verlag, London, UK (1978)
15. Woodcock, J., Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)
16. Ochodek, M., Nawrocki, J.: Automatic transactions identification in use cases. In: Balancing Agility and Formalism in Software Engineering, Berlin, Heidelberg, Springer-Verlag (2008) 55–68
17. Abrial, J.R.: Faultless systems: Yes we can! Computer **42**(9) (2009) 30–36

On an Extensible Rule-based Prover for Event-B

Issam Maamria, Michael Butler, Andrew Edmunds, and Abdolbaghi Rezazadeh

ECS, University of Southampton, Southampton SO17 1BJ, UK
{im06r, mjb, ae2, ra3}@ecs.soton.ac.uk

1 Motivation

The Rodin platform [3] provides the practical setting to carry out modelling in Event-B. It seamlessly integrates modelling and proving, and provides an extensible and configurable mechanism that can be adapted to different application domains and development methods [1]. The Rodin platform provides a proving infrastructure that has certain limitations. Extending the prover with proof rules (rewrite and inference rules) requires a certain level of competence using the Java programming language as well as good knowledge of the toolset's internal architecture. A further complication of this approach is that it became non-trivial to verify the soundness of the prover after adding new rules. This paper presents our approach when dealing with prover extensibility and soundness in the context of Event-B.

2 The Theory Construct

Theories will provide a mechanism by which the user can extend the proof capabilities of the Rodin platform by specifying *rewrite* rules. Proof obligations will be generated to verify the soundness of the prover augmented with the new rules. In essence, the theory construct will allow a degree of *meta-reasoning* to be carried out within the same platform in a similar fashion to Event-B reasoning. Figure 1 outlines the structure of the theory construct. In what follows, we briefly describe each of the elements of the theory construct:

1. *Sets*. A theory can define a number of given sets on which it is parametrised.
2. *Metavariables*. A theory can define a number of metavariables each of which has a type.
3. *Rewrite Rules*. Rewrite rules are one-directional equations that can be used to rewrite formulas to equivalent forms. As part of specifying a rewrite rule, the *theory developer* decides whether the rule can be applied automatically without user intervention or interactively following a user request.

3 Rewrite Rules

A rewrite rule defines how a formula lhs may be rewritten to one of several formulae rhs_i provided condition C_i holds. The following two definitions formalise the concept of rewrite rules within theories. Note the use of the well-definedness operator $\mathcal{D}$ [2].

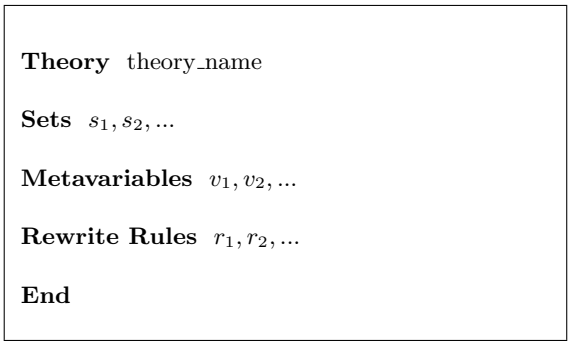


Fig. 1. The Theory Construct

Definition 1 (Rewrite Rule). *A rewrite rule is of the form*

$$\begin{aligned} lhs \rightarrow\ & C_1 : rhs_i \\ & ... \\ & C_n : rhs_n \end{aligned}$$

where:

1. $n \geq 1$,
2. *lhs is not a meta-variable but may contain metavariables,*
3. *lhs and* rhs_i *(for all* i *such that* $1 \leq i \leq n$*) are formulas of the same syntactic class i.e., both expressions or both predicates,*
4. C_i *(for all* i *such that* $1 \leq i \leq n$*) are predicates,*
5. C_i *and* rhs_i *(for all* i *such that* $1 \leq i \leq n$*) only contain free variables from lhs,*
6. *lhs and* rhs_i *(for all* i *such that* $1 \leq i \leq n$*) have the same type if lhs is an expression.*

Note. In this paper, we only consider rewrite rules whose left hand side is a basic predicate (e.g., $\subseteq$) or is an expression not involving binding. More generally, we do not consider rules that require side conditions (i.e., non-freeness conditions).

Definition 2 (Sound Rewrite Rule). *A rewrite rule*

$$\begin{aligned} lhs \rightarrow\ & C_1 : rhs_i \\ & ... \\ & C_n : rhs_n \end{aligned}$$

is said to be sound if the following sequents are valid:

1. $H, \mathcal{D}(lhs) \ \vdash\ \mathcal{D}(C_i)$ *for all* i *such that* $1 \leq i \leq n$,
2. $H, \mathcal{D}(lhs), C_i \ \vdash\ \mathcal{D}(rhs_i)$ *for all* i *such that* $1 \leq i \leq n$,

3. (a) $H, \mathcal{D}(lhs), C_i \vdash lhs = rhs_i$ *for all* i *such that* $1 \leq i \leq n$ *if* lhs *is an expression, or;*
   (b) $H, \mathcal{D}(lhs), C_i \vdash lhs \Leftrightarrow rhs_i$ *for all* i *such that* $1 \leq i \leq n$ *if* lhs *is a predicate,*

where H *is a predicate providing typing information for all free variables occurring in* lhs*.*

The previous definition ensures that rewrite rules are both *validity-preserving* and *WD-preserving*.

4 The Theory Prototype Plug-in

A theory prototype plug-in has been developed as an extension to the Rodin platform. The plug-in offers the following capabilities:

1. Users can develop and validate theories in the same way as contexts and machines.
2. Users can deploy theories to a specific directory where they become available to the interactive and automatic provers of Rodin.
3. Users can use rewrite rules defined within the deployed theories as a part of the proving activity. A pattern matching mechanism is implemented to calculate applicable rewrite rules to any given sequent.

5 Further Work

This work can be extended to enable the specification and validation of inference rules within theories. It can also be extended by providing a clear set of guidelines to help the theory developer with deciding whether a rule can be applied automatically. Finally, the verification of the pattern matching mechanism will give more confidence in this approach.

References

1. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *International Conference on Formal Engineering Methods (ICFEM)*, 2006.
2. Jean-Raymond Abrial and Louis Mussat. On using conditional definitions in formal theories. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 242–269, London, UK, 2002. Springer-Verlag.
3. Michael Butler and Stefan Hallerstede. The Rodin Formal Modelling Tool. *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London.*, December 2007.

Improving traceability between KAOS requirements models and B specifications: Feedback on a localization component

Abderrahman Matoussi[1] and Dorian Petit[2]

[1] University Paris-Est
Laboratory of Algorithmics, Complexity and Logic
Créteil, France
abderrahman.matoussi@univ-paris12.fr
[2] University Lille Nord de France, F-59000 Lille,
UVHC, LAMIH, F-59313 Valenciennes
CNRS, UMR 8530, F-59313 Valenciennes, France
dorian.petit@univ-valenciennes.fr

Abstract. In this paper we give feedback about the B specification of a localization component. We discuss the benefit of using a KAOS modeling to obtain the architecture of the B specifications.

1 Context

A localization system is a critical part of a land transportation system. Many positioning systems have been proposed over the last years. GPS, one of the most widely used positioning system, is perhaps the best-known. This system belongs to the GNSS (Global Navigation Satellite Systems) family which also regroups GALILEO or GLONASS.

Positioning systems are often dedicated to a particular environment; the GNSS technology, for example, generally does not work indoors. To resolve these problems, numerous alternatives relying on very different technologies have arisen. Those last years, Wireless LAN such as IEEE 802.11 networks have been considered by numerous location systems. These systems all use the radio signal strength to determine the physical location.

Localization systems can therefore be designed using various technologies like wireless personal networks such as Wifi or Bluetooth [3, 4], sensors [5], GNSS repeaters or visual landmarks. The aim of this paper is to give some feedback on an ongoing work on the specification of a localization software component that used GPS, Wifi and sensors technologies. We will just focus on the architecture of the specifications and how a KAOS model help us to build it.

The remainder of the paper is organized as follows. Section 2 overviews the first B specification that we have developed and what are the problems we have encountered. Section 3 presents the KAOS methodology and the goal model that we propose for the localization component. Then, Section 4 explains how we derive a new B architecture specification. Finally, Section 5 discusses the benefits of such approach and presents the future works.

2 The first B specification

Once we have defined the different properties which we want to consider in the specification, we began to develop the B specifications [1]. Figure 1 presents the first and simple architecture that we have obtained. The main difficulty when we develop a localization component is to find the correct algorithm that merges positioning data. We planned to refine the *Location* machine to be more and more precise in the manner to do that. The main problem was to take into account all the properties we have to deal with. At this stage, we think that a semi formal model will be very useful in order to have guidelines on how to do.

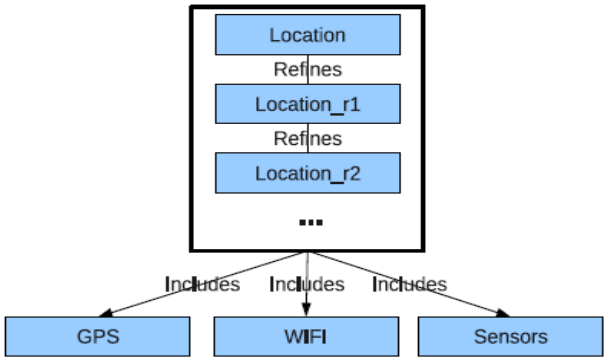


Fig. 1. The first architecture of the B machines

3 The KAOS model

KAOS (Knowledge Acquisition in autOmated Specification) [2] is a goal-based requirements engineering method. KAOS requires the building of a data model in UML-like notation. The particularity of KAOS is that it is able to implement goal-based reasoning. A goal defines an objective the system should meet, usually through the cooperation of multiple agents such as devices or humans. KAOS is composed of several sub-models related through inter-model consistency rules. The central model is the *goal model* which describes the goals of the system and its environment.

In this paper, we have modeled a localization component based on several basic and off-the-shelf technologies : GPS, WIFI and sensors. This localization system will be used in a model of an autonomous vehicle : a CyCab. Figure 2 shows a KAOS goal model of the localization component. Each Goal is described informally in natural language.

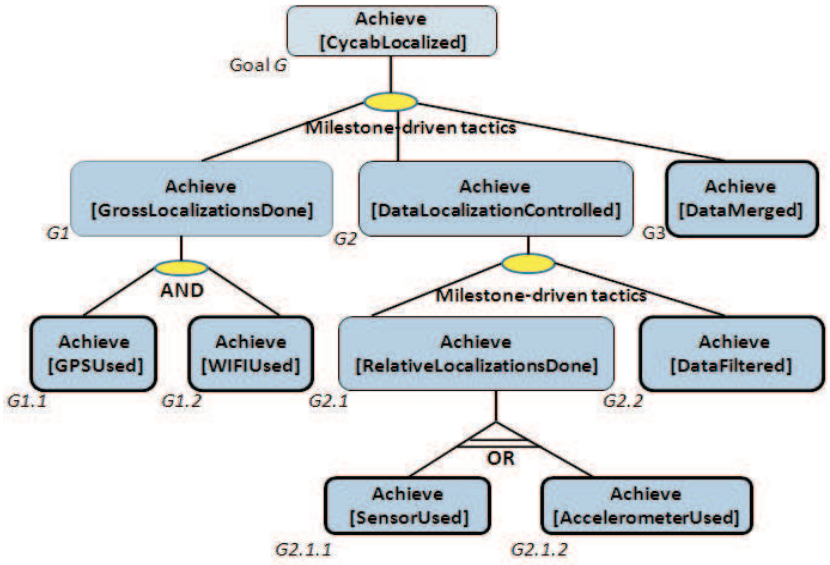


Fig. 2. KAOS goal model of a localization component

4 From KAOS to B

We did not have enough space to describe how we transform the KAOS model into B specifications, the interested reader can refer to [6] for more details.

The main idea is to specify a correspondence rule between each concept of the goal model and B elements. Up to now, we consider only functional goals of type *Achieve* [2]. A B machine is associated to each goal. This machine contains an operation that "realizes" the goal; i.e. it describes the "work" to perform to reach the goal, in terms of generalized substitutions. The refinement of a goal is represented by a B refinement machine that refines the machine; the abstract operation is refined by a concrete one. This operation is built by combining operations of the machines that correspond to the sub-goals of the more abstract goal and are included in the B machine via the inclusion relationship. The nature of the combination depends on the goal refinement pattern (Milestone, AND, OR).

Figure 3 shows the structure of the different B machines. The obtained architecture allows us to easily establish traceability links between KAOS goals and the B operations that realizes these goals.

5 Conclusion and future works

The specification of a localization component brings us to use the KAOS goal-based requirements engineering method. Since goals play an important role in

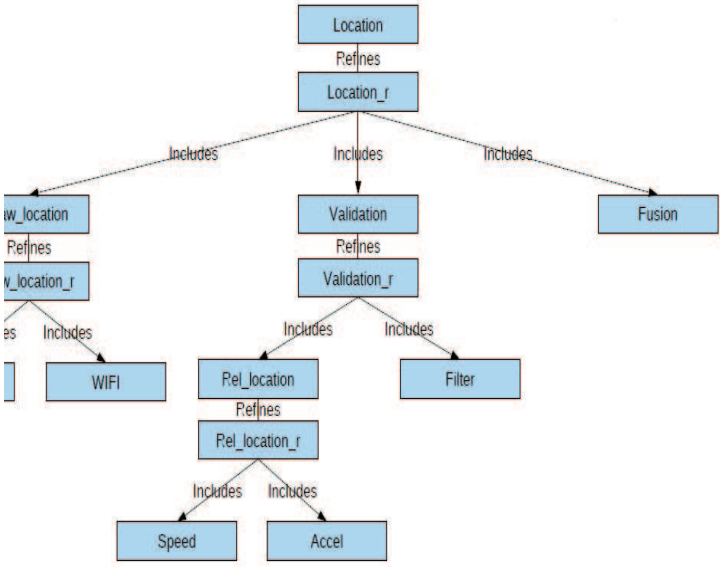


Fig. 3. The architecture of the B machines

requirements engineering process, the proposed mapping comes down to ensure traceability between KAOS goal models and B machines. As a consequence, rather than establishing traceability from the KAOS requirements model as a whole, we propose to establish traceability from individual goals that are part of the KAOS goal model. The main contribution of our approach is that it establishes the first brick toward the construction of the bridge between the non-formal and the formal worlds as narrow and concise as possible. This brick balances the trade-off between complexity of rigid formality (B method) and expressiveness of semi-formal approaches (KAOS). Furthermore, by discharging the proof obligations generated by the B refinement process, we can prove some properties of consistency on the goal model. The current work is still partial and we are actively working on its extensions. Regarding the different KAOS goal model concepts, we need now to consider the translation of the concepts of domain properties and non functional goals. We plan also applying the approach on a number of case studies. At tool level, we plan to develop a connector that establish a partial automated traceability between KAOS goal models and B specifications.

6 Acknowledgements

The work in this paper is partially supported by the TACOS project ANR-06-SETI-017 founded by the french ANR (National Research Agency). We would also like to thank Professor Rgine Laleau for his valuable comments and fruitful discussions that helped a lot for the improvement of this work.

References

1. J.R. Abrial. The B-Book: Assigning programs to meanings. Cambridge University Press, 1996.
2. A. van Lamsweerde. Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, 2009.
3. J. Hallberg and M. Nilsson and K. Synnes. Positioning with bluetooth. In *10th Int. Conference on Telecommunications (ICT'2003)*, pages 954-958, 2003.
4. J.A. Royo and E. Mena and L.C. Gallego. Locating Users to Develop Location-Based Services in Wireless Local Area Networks. In *Symposium on Ubiquitous Computing and Ambient Intelligence (UCAmI2005)*, pages 471-478, Granada, Spain, 2005.
5. R.J. Orr and G.D. Abowd. The smart floor: A mechanism for natural user identification and tracking. In *Conference on Human Factors in Computing Systems (CHI2000)*, pages 1–6, 2000. ACM Press
6. A. Matoussi and R. Laleau and D. Petit. Bridging the gap between KAOS requirements models and B specifications. In *Technical Report TR-LACL-2009-5, LACL, University of Paris-Est (Paris 12)*, *http://lacl.univ-paris12.fr/Rapports/TR/TR-LACL-2009-5.pdf, 2009*

Author Index