Modelling and Analysing the Behaviour of Software

James R. Williams

Department of Computer Science,
University of York, UK
jw@cs.york.ac.uk

Abstract. Developing models of software is becoming more important as the size and complexity of software increases dramatically. The ability to analyse and verify these models is paramount to ensuring that the software is of an acceptable quality. This paper presents work in progress for the definition of a new language and tool support to aid the specification and analysis of software.

1 Model-Driven Software Development

Models have been used in many engineering disciplines for years. In software engineering, however, models have often played a secondary role – as documentation or a means of problem exploration. *Formal methods* focus on rigorous specification and analysis in order to improve the quality of software produced. *Model-driven engineering* (MDE) is another, complementary, approach to developing software, that treats models as first-class artefacts in the development process. Models are used to specify a system at the level of the application domain, and through a series of automatic transformations, generate code. Models in MDE conform to a *metamodel* – another model which describes and constrains the form a model can take. MDE emphasises automation and enables scalable design through decomposition, but often lacks rigorous semantics and has few analysis techniques. The *Eclipse Modeling Framework* (EMF) [11] provides tool support for the key concepts of MDE – creating models and using those models to generate code – and is arguably the most widely used modelling framework at present.

2 Modelling and Analysing Behaviour

Formal methods offer different approaches to specifying software and systems. In order to specify behaviour, (state-based) languages such as Z, B and Alloy, allow the specification of *operations* to add behaviour to their state descriptions. Alternatively, (event-based) languages such as Promela and SAL relate more closely to automata and describe behaviour by defining explicit *transitions* between states.

In MDE, models are treated as first-class citizens, yet there is a less formal model*ling* process than in the formal methods world. EMF focuses on modelling the structure of software, using UML-esque class diagrams. OCL [9] complements EMF models (and UML class diagrams) by providing the ability to add formal constraints to classes and operations, but cannot, however, describe the complete dynamic behaviour of a class diagram. There has been work on simulating and animating EMF models [3, 10], but these approaches focus on generating tools to animate specific models and require substantial amounts of customisation and extra specification. Furthermore, they don't perform any formal analysis, such as invariant checking, on the models and the simulation is not exhaustive.

Semi-formal models, such as UML models, are often formalised into formal languages in order to make use of the rigorous analysis tools (see, for example, [12, 2]). A transformation to a different language, however, can add a large overhead. The transformation needs to be proven to preserve the semantics of the source model, and certain aspects of the source model may not map well to the target language. Furthermore, there is a need to determine backward traceability between the source and target language. If an error is found in the target language, this must be mapped back to the source, and any error messages must be transliterated to make sense with respect to the source. Alternatively, we require that the user be familiar with both the source and target language, and the target tool support in order to address any issues.

3 The Model Behaviour Language

We are developing the *Model Behaviour Language* (MBL), a language for the specification and analysis of the structure and behaviour of software, that is integrated with current state-of-the-art practices in the field of MDE, and utilises rigorous analysis from the field of formal methods. The design of MBL has started with a core set of features for defining behaviour that shall be extended and refined as the language evolves through practical experience.

3.1 Motivation

MBL aims to provide a way to rigorously describe the behaviour of a system, in a manner that is tightly coupled with the system's structural definition and that integrates with the current best-practices in model-driven engineering. As discussed in section 2, there are few ways in which behaviour can be described in MDE, namely OCL and the UML interaction diagrams. MBL introduces support for behaviour modelling, by tightly coupling behavioural specifications with a commonly used structural definition notation.

Due to its de facto status in MDE, MBL is used to describe the behaviour of EMF models. EMF alone provides only the ability to specify class structure, and has no support for behavioural definitions. EMF supports only class modelling, therefore MBL extends EMFs structural modelling to include behaviour modelling and automated analysis. As opposed to the approaches to simulating

models mentioned in section 2, MBL requires no code generation, or extra information to be specified – the model is simulated directly from the specification, and the simulation is exhaustive.

The aim of MBL is not only to allow developers to think about the behaviour of their software, but also to analyse and reason about the specified behaviour to gain confidence in their design.

3.2 Language Overview

MBL is composed of three sub-languages that work together to specify the system:

State description language This language is used to describe the structure of the physical elements in the specification. The state description language mimics the style of *Emfatic*[1] – a popular textual language used to define and generate EMF metamodels. Using Emfatic to define the state will reduce the entry cost to using MBL, as many metamodellers are already familiar with the notation.

Event language The state is extended with behavioural definitions through the introduction of *events*. Events introduce behaviour to the models, and are structured like Hoare triples – having applicability rules in the form of a *precondition*, a *body* section and any guarantees declared in the *postcondition*.

Expression language This language is used to specify the actions taken when an event fires, and any pre- or postconditions that event may have.

3.3 Worked Example

The language is introduced by modelling a simple problem – specifying a simple air traffic control system for an airport. The (partial) MBL solution is below.

```
1  package AirportExample;
2
3  class AirportScenario {
4     val Airport[*] airports;
5  }
6
7  class Aircraft { }
8
9  class Airport {
10    val Aircraft[*] landed;
11    val Aircraft[*] permission;
12    inv RestrictNumberLanded: landed.size() <= 20;
13
14    event givePermissionToLand(Aircraft a) [  ] {
15       permission.add(a);
16    } [  ]
```

[1] Emfatic web page: http://wiki.eclipse.org/Emfatic

```
17
18      event planeLands(Aircraft p) [ permission.includes(p) ]{
19              landed.add(p);
20      } [ ]
21  }
```

The state description closely mimics Emfatic's syntax. We model a container class, AirportScenario, to store the set of airports in our system – defined using the Emfatic keyword val and the one-to-many multiplicity ([*]), meaning that each referenced object can only belong to the referencing class.

There are two classes of object that need modelling for this problem – aircraft and airports. Airports have two features – landed and permission – which are also defined to be "by value" collections – implicitly adding the restriction that an aircraft can only be landed at one airport. In languages like Alloy and Event-B, an extra constraint is required to enforce this.

The MBL keyword inv is used to define class invariants (something not possible with pure Emfatic). Here, we define an invariant to restrict the number of aircraft that can be landed at any one time.

The state of the system is affected (or observed) using the notion of *events*. Events are abstract representations of actions that can occur in the system, and do not directly map to the structure of the model in the way that a class *operation* would. Events can later be *refined* (see section 3.5) by operations. During the analysis stage (see section 3.4), events and operations make up the transitions between states.

Preconditions to an event occur in the first set of square brackets, and post-conditions in the second set. An event has a name and can include any number of parameters. In the givePermissionToLand event, there is one parameter – the Aircraft that we wish to grant permission to. The expression language offers us the add method for collections, but also generates an implicit precondition that the aircraft a is not already a member of the permission collection.

To analyse the specification, we can define an initial state. An initialisation is given a name to allow for multiple initialisation declarations. The model is described using the OMG's Human Usable Textual Notation (HUTN) [8], a textual syntax for writing models, designed to be user-friendly as models are usually stored in XML. An example HUTN model that contains one airport with one aircraft landed and two with permission is below.

```
22  init "InitFromHutn" : AirportScenario {
23      airports: Airport "Airport1" {
24          landed: Aircraft "Aircraft1" { }
25          permission: Aircraft "Aircraft2" { },
26                      Aircraft "Aircraft3" { }
27      }
28  }
```

As shown, the root node is given context using the colon operator, similar to the way in which OCL declares contexts.

This toy example is for illustration only. We are currently working on larger examples, including the classic steam boiler control problem for comparison with existing solutions found in [1].

3.4 Analysis

We are currently implementing tool support for MBL in the form of an Eclipse-based editor and a model checker for simulating and checking properties. The MBL grammar is implemented using Xtext[2], a language for defining concrete and abstract syntaxes simultaneously with tool support to generate a fully featured Eclipse based editor.

As mentioned in section 2, specifications are often transformed into other languages to make use of analysis tools. To avoid the issues of mapping to another language, MBL models will be analysed directly, without transformation, by making use of a generic model checker.

In 2008, de Jonge [4, 5] developed a Java implementation of the SPIN model checker, *SpinJa*, which aimed to implement the core of SPIN in a reusable and extendable manner, without suffering too much of a performance degradation. The design of SpinJa conforms to Kattenbelt's [6] conceptual framework for unifying model checking to support the reuse of algorithms. The framework consists of three layers – the first two define verification and simulation algorithms, and abstractly describe the model being checked. The third layer tailors the model to the target language – in SpinJa's case, Promela.

MBL will provide a custom top layer, replacing SpinJa's Promela layer, whilst making use of the algorithms provided by SpinJa's abstract layers. Whereas SpinJa (and SPIN) compiles a Promela specification to Java (or C) and executes the resulting program, MBL will simulate the model directly in memory. This does mean, however, that an event's preconditions, postconditions and body need to be interpreted. Rather than developing a bespoke interpreter for MBL, we will allow expressions to be written in an existing language, the *Epsilon Object Language* (EOL) [7] – a language that allows querying and updating models – and make use of EOL's interpreter to evaluate expressions.

3.5 Future Work

The aim of MBL is to aid, not only the specification of software, but also its creation. MBL will be able to support data refinement and event refinement, as well as couple with EMF's code generation features to implement the body of class operations. Specifications, along with results from analysis, will also be used to generate test cases for the final implementation to improve confidence that the implementation meets the specification. Work is underway to develop an Eclipse based interactive simulator, similar to that of ProB's[3]. Finally, we plan to extend MBL to support the specification and verification of temporal properties.

[2] Xtext project website: http://www.eclipse.org/Xtext/

[3] ProB animator and model checker: http://www.stups.uni-duesseldorf.de/ProB/

4 Summary

This paper has introduced the Model Behaviour Language, a language designed to aid the specification and analysis of software, with a focus on behaviour. MBL integrates with the current state-of-the-art technologies in the field of model-driven engineering, providing code and test-case generation as well as simulation and analysis. MBL seamlessly integrates formal analysis into MDE best practices, analysing the model directly, rather than a representation of it.

References

1. J-R. Abrial, E. Borger, and H. Langmaack. Formal methods for industrial applications: Specifying and programming the steam boiler control. In *Lecture Notes in Computer Science*, volume 1165, October 1996.
2. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, volume 4735, pages 436–450, Nashville, USA, 2007. Springer.
3. Xavier Crégut, Benoit Combemale, Marc Pantel, Raphael Faudoux, and Jonatas Pavei. Generative technologies for model animation in the TopCased platform. In T. Kühne et al., editor, *6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, volume 6138 of *Lecture Notes in Computer Science*, pages 90–103, Paris, France, June 2010. Springer.
4. M. de Jonge. The SpinJ Model Checker. Master's thesis, University of Twente, Enschede, The Netherlands, September 2008.
5. M. de Jonge and Theo C. Ruys. The SpinJa model checker. In J. van de Pol and M. Weber, editors, *SPIN 2010*, volume 6349 of *Lecture Notes in Computer Science*, pages 124–128. Springer-Verlag Berlin Heidelberg, 2010.
6. M. A. Kattenbelt, T. C. Ruys, and A. Rensink. An object-oriented framework for explicit-state model checking. In *VVSS 2007*, pages 84–92, Eindhoven, Netherlands, March 2007.
7. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Object Language (EOL). In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture — Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin Heidelberg, 2006.
8. The Object Management Group. *UML Human-Usable Texual Notation*, 1.0 edition, August 2004.
9. The Object Management Group. *Object Constraint Language*, May 2010.
10. Daniel A. Sadilek and Guido Wachsmuth. Prototyping visual interpreters and debuggers for domain-specific modelling languages. In Ina Schieferdecker and Alan Hartman, editors, *4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'08)*, volume 5095 of *Lecture Notes in Computer Science*, pages 63–78. Springer-Verlag, 2008.
11. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, second edition, 2009.
12. H. Treharne, E. Turner, R. F. Paige, and D. S. Kolovos. Automatic generation of integrated formal models corresponding to UML system models. In M. Oriel and B. Meyer, editors, *TOOLS EUROPE*, volume 33 of *LNBIP*, pages 357–367, 2009.