UML-B AND EVENT-B: AN INTEGRATION OF LANGUAGES AND TOOLS

Colin Snook and Michael Butler
University of Southampton,
Southampton,
United Kingdom
{cfs,mjb}@ecs.soton.ac.uk

ABSTRACT
UML-B is a graphical front end for Event-B. It adds support for class-oriented modelling but retains the Event-B concept of a closed system characterized by families of spontaneous events. UML-B is similar to UML but is essentially a new notation based on a separate meta-model. We provide tool support for UML-B, including drawing tools and a translator to generate Event-B models. The tools are closely integrated with the Event-B verification tools so that when a drawing is saved the translator automatically generates the corresponding Event-B model. The Event-B verification tools (syntax checker and prover) then run automatically providing an immediate display of problems. We introduce the UML-B notation its tool support and its integration with Event-B.

KEY WORDS
Visual modelling languages, Formal, UML, Event-B

1. Introduction

UML-B is a graphical formal modelling notation based on UML [1] and relies on Event-B [2] and its verification tools [3]. UML-B and Event-B have been developed under the Rodin project, which is an EU framework 6, STREP project [4]. In previous work [5] we developed a specialisation of the UML called UML-B using the profiling extension mechanism included in UML. That, initial, version of UML-B was based on the Rational Rose UML modelling tool [6] and translated into, so called, 'classical B' [7] (B before Event-B). However, the degree of integration between the tools was poor and unidirectional. The new version of UML-B is implemented in Eclipse [8], is platform independent and closely integrated with the Event-B tools. UML-B is an extension to the Event-B platform and the U2B translator runs as an Eclipse 'builder' so that Event-B models are generated and analysed automatically as soon as the UML-B model is saved. Problems discovered by the verification tools will be fed back and displayed on the UML-B diagrams.
Experience with the initial version of UML-B indicated that the richness and semantics of UML could be misleading for modellers. UML-B used a small subset of UML features, useful for translation into B. However, users were confused over which features they should use and sometimes found that the semantics of UML-B were not quite the same as UML. For our initial attempt at the new UML-B we used the profile and stereotype mechanisms of UML 2.0 This improved UML-B but there was a strong feeling that the profile was an add-on and not an integral part of the notation. There was still the problem that the main notation contained a lot of unused redundant modelling concepts. The profile extension mechanism is intended to be used when a relatively small adaptation of UML is required. When the specialisation is more extensive a new meta-model should be defined. The advantage of defining UML-B via an independent meta-model is that it can be designed to the requirements rather than adapting something more general. Hence UML-B is now a 'UML-like' formal modelling language rather than a specialisation of the UML.

2. Overview of UML-B

The new UML-B provides a top-level Package diagram for showing the relationships between components (machines and contexts) in a project. [Here we adopt the Event-B terminology where a context is the static data (sets and constants) in which the behavioural model is couched. Hence a context configures a model to a particular scenario rather than describing its environment.] As in UML, the package diagram provides a structuring of the model into a collection of contexts. However, UML-B introduces the new concept of refinement, where model complexity is managed by introducing more detailed versions of a machine.
Contexts are described in a context diagram which is similar to a class diagram but has only constant data represented by ClassTypes (equivalent to classes), attributes and associations. Axioms (given properties about the constants) and Theorems (assertions requiring proof) may be attached to the ClassTypes.
Similarly, Machines are specified by a Class diagram. The associations and attributes represent variables and events may be attached to the class to describe how those variables change. Events can also be represented by the transitions on a state machine. The state machine

represents a variable whose type is the set of states in the state machine and transitions are guarded so that they are only enabled when the state machine variable is at their source state. As the transition fires it assigns the target state to the variable. Additional guards and actions concerning other class attributes can be attached to the transition. A class may have several orthogonal state machines similar to orthogonal regions in UML. Each orthogonal state machine represents behaviour in terms of transition events that can interleave since, unless explicitly modelled in the transition guards, the ordering implied by a state machine is independent of any orthogonal state machine. State machines may also be attached to the states of another state machine providing a hierarchical sub-state mechanism, similar to that of UML. Invariants (and theorems) may be attached to classes and states (as well as independently at the machine level).

For textual constraints and actions we use a notation, μB (micro B) that borrows from the Event-B notation. μB differs from Event-B as follows. An object-oriented style dot notation is used to show ownership of entities (attributes, operations) by classes. Variables used in an expression can represent owned features of class instances (such as attributes, associations or state diagrams). The owning instance is specified using the dot notation. For example i.x refers to the value of the variable x belonging to instance i. When an expression is attached to a feature belonging to a class, the owning instance for the current contextual instance is referenced using the reserved word, *self*. The value represented by an expression i.x depends on the cardinality of the variable x. If it is a function a single value is represented; if not, a set of values (corresponding to the relational image) is represented. Since it is often useful to concatenate several of these traversals, we may wish to traverse an association from a set of instances that were returned by a previous expression. For this case, the dot symbol is replaced by a right facing triangle ►.

To give a flavour of UML-B, consider the specification of the telephone book in Fig. 2.1. The classes, NAME and NUMB represent people and telephone numbers respectively. The association role, pbook, represents the link from each name to its corresponding telephone number. Multiplicities on this association ensure that each name has exactly one number and each number is associated with, at most, one name. The properties view shows μB conditions and actions for the add event. The add event of class NAME has the constructor property set (not shown in Fig. 2.1) which means that it adds a new name to the class. It non-deterministically selects a numb, which must be an instance of the class, NUMB, but not already used in a link of the association pbook (see μB guard), and uses this as the link for the new instance (see μB action). The remove event (which is a destructor of class NAME) has no μB action; its only action is the implicit removal of self from the class NAME. This specification is equivalent to the Event-B model shown in Fig. 2.1 and indeed the U2B tool automatically produces the Event-B model from the UML-B version.

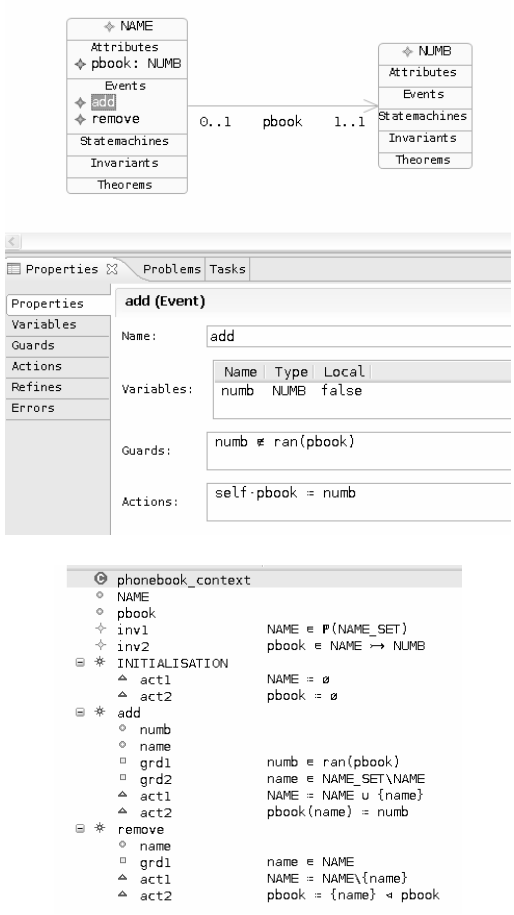

Fig. 2.1. UML-B Specification of a phone book and its equivalent Event-B specification

3. Functionality of UML-B

The UML-B modelling environment consists of a UML-B project containing a UML-B model. A builder is associated with the project so that it runs automatically whenever resources (files) are saved in the project. Four interlinked diagram types (package, context, class and state machine) are provided. The top-level package diagram is opened with an empty canvas by the model creation wizard. This canvas represents the UML-B project. Other diagram types are linked and opened via model elements as they are drawn on the various canvases.

3.1 Package Diagrams

Package Diagrams are used to describe the relationships between top level components (machines and contexts) of a UML-B project. The diagram shows the refinement relationships between Machines, the extension relationships between Contexts and which contexts are seen by each machine. Fig. 3.1 shows an example of these relationships between two machines (m1 and r1) and two contexts (cx1 and cx2). Notice the properties view at the

bottom of the perspective. This is where the property details of model elements are configured. In this case context cx1 is selected in the drawing and the properties view contains a button to open the context diagram for cx1.

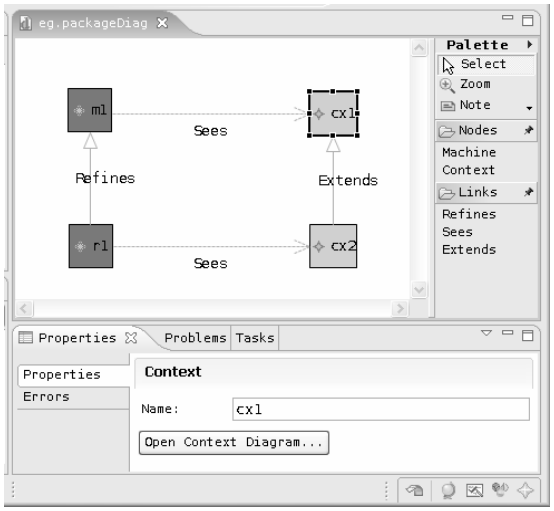

Fig. 3.1. Example package diagram

3.2 Context Diagrams

The Context diagram defines the static (constant) part of a model. ClassTypes define 'given' sets and constant attributes that are based on them (i.e. lifted). For example, Fig. 3.2 shows a ClassType PERSON that has an attribute, id. Properties provide control over the cardinality features of id. In this case, we wish all instances of PERSON to have exactly one id and for that person's id to be unique. Hence, the functional, total and injective properties are true. (Functional and total are defaults for attributes). Fig. 3.2 also shows the corresponding Event-B context that has been produced by the U2B tool.

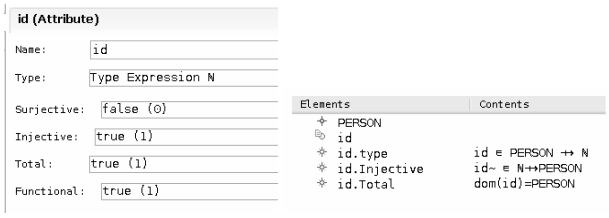

Fig. 3.2. Properties view of an attribute and its Event-B translation

Fig. 3.3 shows further features of the context diagram. An association, accounts, provides a constant in the same way that the attribute does. The only difference is that associations do not default to functional and total. Hence, UML-B simplifies the treatment of associations compared to UML since UML-B associations are always uni-directional and contained by the source whereas UML associations may be uncontained independent elements that are referenced by the two roles belonging to the two classes involved. This simplification makes translation into Event-B easier by removing the full flexibility of UML associations. However, since the UML approach would require multiple redundant variables which is undesirable when proving consistency, the loss of flexibility is justified.

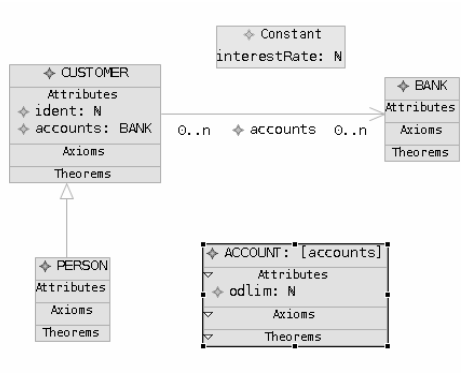

Fig. 3.3. Context diagram showing ClassTypes

The ClassType, PERSON has the ClassType CUSTOMER as its superset. Hence it is translated into a constant which is a subset of the given set, CUSTOMER. The ClassType, ACCOUNT has its *instances* property set to accounts. The *instances* property provides a means to model a ClassType from a set of instances defined elsewhere within the context. In this example ACCOUNT is the set of link mappings in the association, accounts. Hence, ACCOUNT corresponds to a UML association class. This mechanism is more flexible than association classes since any expression resulting in a set can be used.

3.3 Class Diagrams

The class diagram is used to describe the behavioural part of a model. Classes represent subsets of the ClassTypes that were introduced in the context. The class' associations and attributes are similar to those in the context but represent variables instead of constants. For example, in Fig. 3.4, the bank class has an association, accounts, with the account class which will be translated into a variable, accounts, of type bank $\leftrightarrow$ account and initialised to $\varnothing$. Additional invariants giving the functional nature of the inverse relation and coverage of the range, reflect the 1..1 cardinality at the source end of the association. The attribute, balance, of class, account, defaults to a total function. A class invariant specifies that the account's balance must be greater than its overdraft limit, odlim.

The correspondence between an association's multiplicity constraints (introduced in Fig. 3.3, but also applicable to associations between classes) and the constraints on the resulting Event-B relationship is clear from the drawing tool. The multiplicity properties are described using the usual mathematical terminology (functional, total, injective, surjective) with the UML style multiplicity also shown and annotated automatically on the diagram. The correspondence of an association multiplicity, a..b $\rightarrow$ c...d

is as follows, a=1 : surjective, b=1 : injective, c=1 : total and d=1: functional.

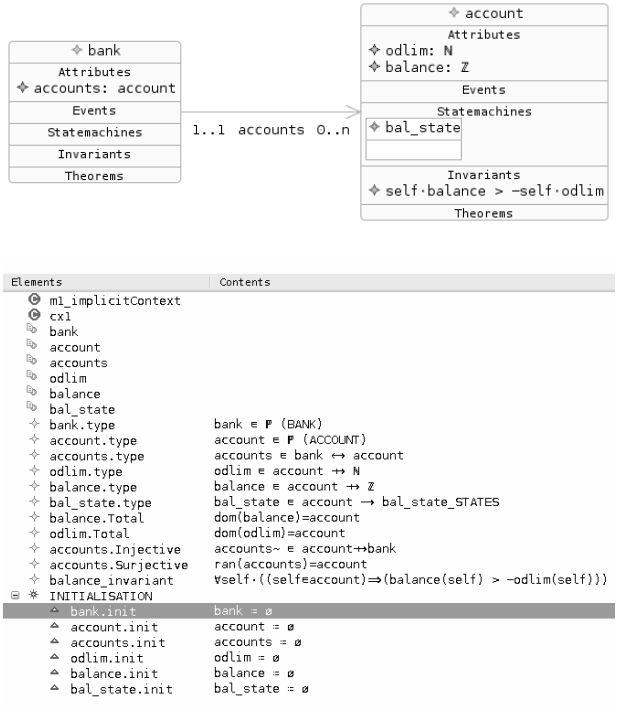

Fig. 3.4. Example class diagram and its Event-B translation

Classes may contain events that modify their attributes. An example was shown in the introduction (Fig. 2.1). Such events implicitly utilise a local variable that non-deterministically selects an instance of the class. This instance is referred to via the reserved word self when referencing the attributes of the class. Constructors and destructors add or remove instances from the current instances of the class. We have also found it useful to model 'fixed' classes where instances cannot be added or removed. Many systems (e.g. embedded systems) have this static configuration. The use of generic UML-B models to verify and validate complex static configurations is investigated in [9].

3.4 State Machine Diagrams

State machines attached to classes represent a variable of the class that partitions the behaviour of the class in some way. For example, the state machine, bal_state, of class, account, partitions the behaviour of the account class into two states, black and red (Fig. 3.5). The transitions of a state machine represent events with the additional behaviour associated with the change of state implied by the transition. That is, the event can only occur when the instance is at its source state and, when it fires, changes the state of the instance to the target state. In the previous version of UML-B, the transitions represented branches of a select statement within an event and all the transitions with a similar name were collated into a single event. With Event-B this is no longer possible since all the selection constructs have been removed. Instead, events are selected when their guards are true. Hence each transition represents a separate event. As with events, event variables can be added to the transition to provide a non-deterministically chosen value to be used in the transition's guards and actions.

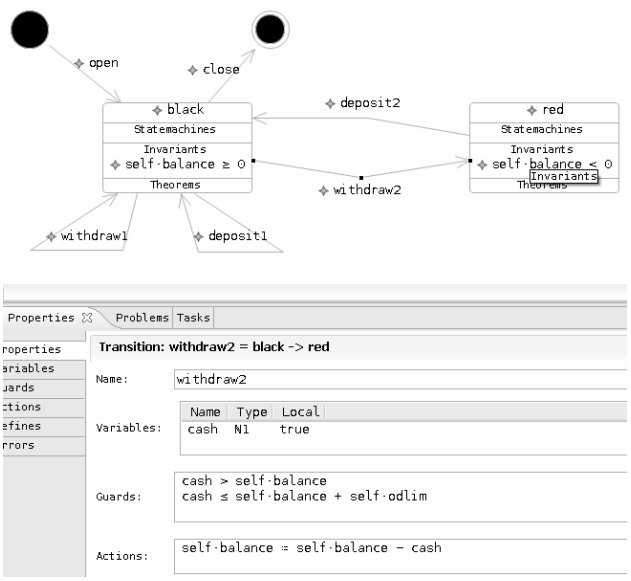

Fig. 3.5. Example statemachine diagram showing ancillary properties of a transition

In order to define the type of the state variable, bal_state, the translation needs a given set that consists of the two states, black and red. This is defined in an 'implicit' context for m1 as shown in Fig. 3.6.

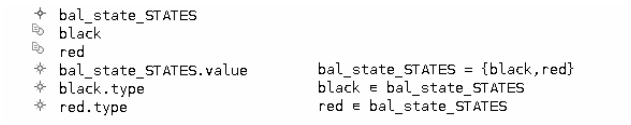

Fig. 3.6. Translation of statemachine into Event-B (data parts)

Invariants may be attached to the states as shown in Fig. 3.5. During translation, these invariants are universally quantified over the class instances and constrained by an antecedent giving the owning state, in this case, bal_state=red. This provides an efficient mechanism for linking the meaning of the states to other class variables.

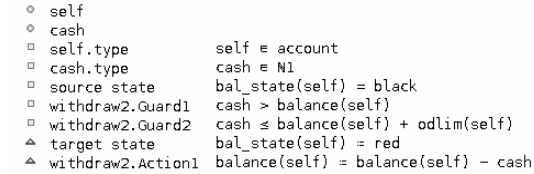

Fig. 3.7. Translation of statemachine transition into Event-B

The translation of a transition (for example, withdraw2 is shown in Fig. 3.7) is similar to class events except that a guard for the starting state (source.state) and an action to move to the target state (target.state) are added.

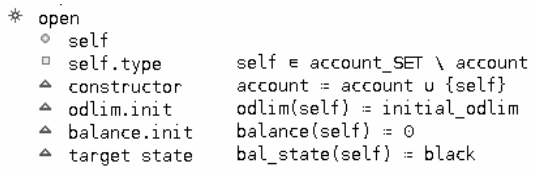

Fig. 3.8. Translation of statemachine constructor into Event-B

The transition from the starting state defines a constructor for the class. Hence the translation of this transition (Fig. 3.8) selects an unused instance and adds it to the set of current instances and initialises all the class variables for that instance. Similarly, the transition to a final state is a destructor and removes the instance from the current instances and from the domain of all the class variables.

An alternative, semantically equivalent, translation of state machines is provided and can be selected per state machine by setting a property switch in the diagram. In this alternative translation a variable is provided for each state which represents the instances currently in that state. The choice of translation is influenced by the model. For example, the alternative translation is useful when the transitions are guarded by the number of other instances in particular states, since it is then convenient to refer to the cardinality of a state.

4. Implementation of UML-B

The abstract syntax of the UML-B language is given by a metamodel. The metamodel is a precise description of the abstract syntax of the UML-B language and is used to automatically generate repository and editing utility code. Fig. 4.1 shows part of the UML-B metamodel. Italicised classes represent abstract meta-classes. A base class, *UMLBelement*, provides a name and error marking scheme. *UMLBconstrainedElement* represents *UMLBelement*s that own constraints (axioms or invariants) and theorems. Note that the metamodel does not define the syntax of predicates, merely representing them as a string attribute of the **UMLBPredicate** class. One subtype of *UMLBconstrainedElement* is subtyped via *UMLBconstruct* into **UMLBMachine** and **UMLBContext**, which own containments of **UMLBClass** and **UML-BClassType** respectively. Fig. 4.1 omits many features such as state machines, variables and events that are contained within the metamodel.

In some cases, where constructing a fully constrained graphical model is not possible, OCL constraints [10] are added to the model. An example is the metamodel for states and transitions where a state that has the initial attribute set is not allowed to have incoming transitions. It may have been possible to subclass states in some way so that initial states were prevented from having incoming transitions. However, it was felt that a graphical depiction of this situation would complicate the model. OCL constraints are either implemented within the graphical modelling tool to prevent invalid models being created or are used in a pre-translation validation stage to ensure that the model is well-formed.

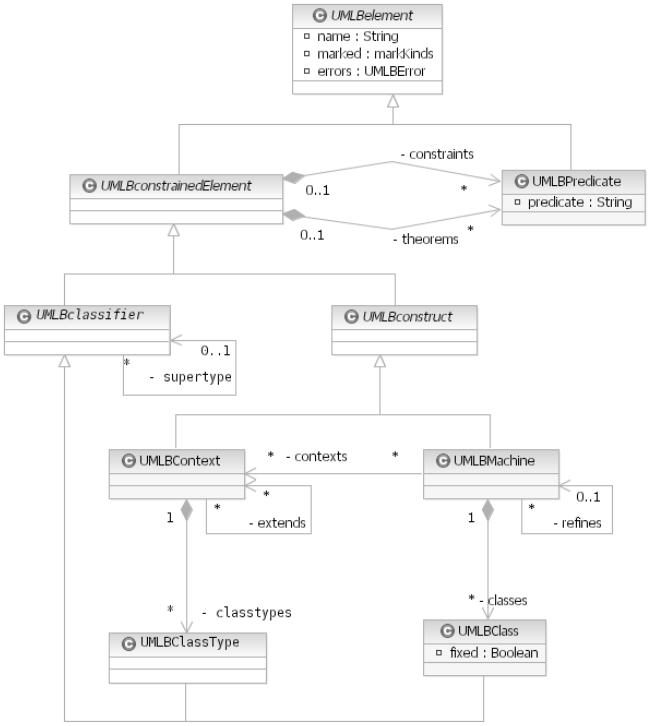

Fig. 4.1. The UML-B metamodel (part of).

The EMF (Eclipse Modelling Framework) [11] is an Eclipse project that automatically generates code for a model repository, model editor and API utilities based on an metamodel. The EMF generated code provides utilities to programmatically create and manipulate instances of the metamodel with serialisation provided in XMI. The GMF (Graphical Modelling Framework) [12] is another Eclipse project that, after configuration, will automatically generate code for a graphical modelling tool based on an EMF model.

The UML-B metamodel was imported into EMF in order to generate the Eclipse plug-ins to support the UML-B modelling language. The GMF was then used to generate the UML-B graphical modelling tool. Drawings created using the UML-B modelling tool are saved as serialised UML-B model files. An Eclipse 'builder' responds to changes to such model files and translates them into a Rodin Event-B project. When the Event-B project is saved, the Rodin verification tools (also Eclipse builders) automatically verify the Event-B model and report any errors. A final stage (currently in progress) is to listen for these errors and annotate the UML-B diagrams so that a user can work entirely in the UML-B environment and benefit from the powerful static verification and prover technology provided by Rodin Event-B. It is still expected that there will be proof obligations where the prover requires human assistance to discharge. This requires the modeller to switch perspective to the Event-B prover environment and to work in the Event-B notation. However, one of the primary goals of Event-B is to

achieve better rates of automatic proof so that these instances are reduced.

5. Experience

UML-B has been used to model a failure management system (FMS) [9, 13]. The FMS is a wrapper layer that protects an engine control algorithm from failures in the transducers and machinery it controls. The FMS detects and filters out transient failures. It also monitors transient faults and if persistently occurring declares the transducer as faulty, reverting to an alternative or degraded mode. The models cover several levels of refinement and are fully proven using the Event-B prover. For example, the abstract model describes the detection of persistent transients failures non-deterministically as a pass-fail sequence. This is proven to be refined by a counter mechanism that is used in the implemented system.

A major concern in the FMS project was reducing the semantic gap between specification elements and the problem domain. If the constructs in the notation map easily onto concepts in the problem domain, it is easier to construct and understand descriptions. Object-oriented notations are good at achieving this when the problem domain involves large collections of similar objects with minor variations and plentiful interrelationships. In the FMS case study we used UML-B to specify the generic problem domain in an entity-relationship style that could be instantiated with specification objects to 'configure' the specification for a particular application. UML-B was found to be very suitable for this kind of problem where a generic model is required for instantiation to particular applications.

6. Conclusion

UML-B is a fully integrated graphical front end for Event-B. UML-B has similarities with UML that make it approachable for users that are familiar with UML. The advantages of moving to a completely new metamodel are that the language is more concise for expressing Event-B modelling concepts. The full expressivity of UML is largely redundant for our needs and this can confuse users. UML-B retains sufficient commonality with UML for the main goals of approachability to be attained by industrial users.

Since UML-B automates the production of many lines of textual B, models are quicker to produce and hence exploration of a problem domain is more attractive. This assists novices in finding useful abstractions for their models. We have found that the efficiency of UML-B and its ability to divide and contextualise μB expressions, assists novices who would otherwise be deterred from writing formal specifications. Furthermore, the new event oriented UML-B with its strong integration with the Event-B platform is gaining acceptance as a useful visual aid for experienced formal methods users.

Several groups have investigated UML based graphical renderings of B [14, 15] as well as our own previous work [5]. However, our work is unique in providing a link to Event-B and the first to provide a highly integrated link to a strong formal verification system. Our work also differs by defining its own language which has avoided many of the problems highlighted in previous work.

References

[1] G. Booch, I. Jacobson, and J. Rumbaugh, *The unified modeling language - a reference manual* (Addison-Wesley, 1998).

[2] C. Métayer, J.R. Abrial and L. Voisin, Event-B Language, *RODIN Deliverable D7* [Rodin], 2005.

[3] J.R. Abrial, S. Hallerstede, F. Mehta, C. Métayer and L. Voisin, Specification of Basic Tools and Platform, *RODIN Deliverable D10* [Rodin 2005].

[4] Rigorous Open Development Environment for Complex Systems (RODIN) - IST 511599. http://rodin.cs.ncl.ac.uk

[5] C. Snook and M. Butler, Formal modeling and design aided by UML, *ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 15*(1), 2006, 92 – 122

[6] Rational, *Rose extensibility user's guide – Rational Rose 2000e*. Rational Software Corporation. Part Number 800-023328-000 (2000).

[7] J.-R. Abrial, *The B-Book: assigning programs to meanings* (Cambridge University Press, 1996).

[8] J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, P. McCarthy, *The Java developer's guide to Eclipse, 2nd Edition* (Addison-Wesley, 2004).

[9] C.Snook, M. Poppleton and I. Johnson, Rigorous engineering of product-line requirements: a case study in failure management, *Information and Software Technology*, In press (available on-line 26 Oct 2007).

[10] J. Warmer and A. Kleppe, *The object constraint language second edition – getting your models ready for MDA* (Addison-Wesley 2003).

[11] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose, *Eclipse modelling framework* (Addison-Wesley, 2004).

[12] GMF, The Eclipse Graphical Modelling Framework, *http://www.eclipse.org/gmf/*

[13] E.Troubitsyna (Ed.), Final Report on Case Study Development, *RODIN Deliverable D26*, [Rodin 2007].

[14] K. Lano, D. Clark, and K. Androutsopoulos, UML to B: formal verification of object-oriented models, *Proc. Integrated Formal Methods, 4th International Conference, IFM 2004*, LNCS Vol. 2999 Springer, 187-206.

[15] H. Ledang and J. Souquières, Integrating UML and B specification techniques, *Proc. Informatik2001 Workshop on Integrating Diagrammatic and Formal SpecificationTechniques,* 2001.