Verifying Dynamic Properties of UML Models by Translation to the B Method and Toolkit

Colin Snook & Michael Butler

Declarative Systems and Software Engineering Group
Technical Report DSSE - TR-00-12

September 2000-09-08

www.dsse.ecs.soton.ac.uk/techreports/

Department of Electronics and Computer Science
University of Southampton
Highfield. Southampton, SO17 1BJ, United Kingdom.

Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit

Colin Snook and Michael Butler
Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, United Kingdom.
{cfs98r,M.J.Butler}@ecs.soton.ac.uk

Abstract. Formal languages such as the B language, enable the dynamic behaviour of a system to be investigated and verified. B is particularly suitable for this because of its good level of tool support. A model's behaviour can be explored using the animation facilities in the B Toolkit. It's behaviour can be proven to conform to its specified invariants using the proof tools available in the B tools. The equivalence of two alternative expressions of a model could be established using the tool. By expressing invariants and dynamic behaviour suitably in a UML class diagram the model can be translated into B automatically in order to obtain these benefits.

1 Introduction

We are developing a program to convert UML [7] class diagrams into specifications in the B language [1]. The current version is limited in that the only form of inter class relationship translated is aggregation. The translation relies on suitable expression of additional behavioural constraints in the specification of class diagram components. These constraints are described in the B notation. We envisage benefits to UML users from the provision of a rigorous abstract representation of their models and from the assistance that B provides in enabling the dynamic behaviour of a system to be investigated and verified. Conversely, B users will benefit from being able to create and view models in the UML diagrammatic form.

The B Language and Toolkit

The B language is a formal specification notation that has strong structuring mechanisms and good tool support. There are 2 commercial tools for B, Atelier B and the B Toolkit. We have used the B Toolkit for our translation and animation work, and Atelier B for performing proofs. B is designed to support formally verified development from specification through to implementation. To do this it provides tool support for generating and proving proof obligations at each stage of refinement. The B Toolkit also provides animation facilities so that the validity of the specification can be investigated prior to development. To make large scale development feasible, B provides structuring mechanisms to decompose the specification and its subsequent refinements. These are machines, refinements and implementations. We are mainly concerned with specification and therefore machines. Machines allow an abstract state to be partitioned so that parts of the state can be encapsulated and segregated, thus making them easier to comprehend reason about and manipulate. One machine may 'INCLUDE' another machine. If machine A includes machine B, the state of B is visible to A and alterable via B's operations. A weaker form of interfacing between machines is provided by 'USES'. The using machine has only read access to the used machines variables and cannot invoke its operations. A machine may be used by any number of other machines but may only be included by one other machine.

Benefits of Translating UML to B

A B specification can be animated with the B Toolkit to explore the dynamic behaviour of the modelled system. In UML terms this means that operations of an object can be invoked and the B animator will check pre-conditions, and invariants and display the new state of the system in terms of the object's attributes and relationships with other objects.

A class' dynamic behaviour can be proven to conform to the class' invariants. In UML terms this means that the proof tools will provide assistance in proving that no sequence of invocations of an object's operations can produce a resultant state in terms of the class' attributes and relations with other objects that doesn't conform to the invariant. A safety or business critical property of the system could be specified and verified in this way.

Conversion of other UML forms of dynamic specification may be possible. Meyer and Souquieres [4] have already proposed ways of translating OMT models to B including both class diagrams and state charts. This would enable the equivalence of the different views of dynamic behaviour within UML to be investigated.

UML models prepared for translation to B contain verified invariant and method descriptions (constraints) in a rigorous, abstract notation. This improvement to UML specifications is, in itself, a benefit of the translation process. The UML diagram is given a precise semantics as expressed by its equivalent form in the B notation.

U2B Translator

The U2B translator converts Rational Rose[1] UML Class diagrams into the B notation. U2B is a script file that runs within Rational Rose and converts the currently open model to B. It is written in the Rational Rose Scripting language which is an extended version of the Summit BasicScript language. U2B is configured as a menu option ("Export to B") under the File menu of Rose. U2B uses the object oriented libraries of the Rose Extensibility Interface to extract information about the classes in the logical

[11] Rational Rose is a trademark of the Rational Software Corporation

diagram of the currently open model. The object model representation of the UML diagram means that information is easily retrieved and the program structure can be based around the logical information in the class rather than a particular textual format. U2B uses Microsoft Word97[2] to generate the B Machine files via the OLE interface. The Rose Script uses the object oriented document model of Word97 in order to facilitate the creation of the B Machines. Word template files are used to form the basic layout of the Machines.

Translation of Structure and Static Properties

The translation of Classes, attributes and operations is derived from Meyer and Souquieres' [4] proposals for converting OMT to B. A separate machine is created for each class and this contains a set of all possible instances of the class and a variable which represents the subset of current instances of the class. Attributes are translated into variables with their type defined in the invariant clause of the machine as a relation from the current instances to the UML attribute type. Types can be any of the predefined types of B (including BOOL which is a B library type) or another class. If the type is neither of these it will be added as a parameter of the machine. Types can also be a set of any of these by putting POW(typename) as the type in the UML class specification. U2B could easily be extended to cover sequences and other B data structures in a similar manner. A create operation is automatically provided for each class machine. This picks any instance that isn't already in use, adds it to the current instances set, and adds a maplet to each of the attribute relations mapping the new instance to the appropriate initial value. Currently U2B does not create any other (parent) machine. This works with the current limitations of U2B only allowing the aggregation relationship between classes. If generalization and general associations are allowed this would be insufficient and other structures would be required. Other authors [3], [4], [5], [6], [8] have suggested ways of dealing with the translation of these forms of interaction.

Dynamic Behaviour

The dynamic behaviour modelled on a class diagram that is converted to B by U2B is embodied in the behaviour specification of a classes operations and in invariants specified for the classes. These details are specified in a textual format as annotation to the class diagram. In Rational Rose, 'Specifications' are provided for operations (as well as many other elements) and these provide text boxes dedicated to writing pre and post-conditions for the operation. Unfortunately there is no text box for a class invariant. Warmer and Kleppe [10] suggest putting invariant constraints in a note attached to the class, but notes appear to be treated as an annotation on a particular view in Rational Rose and not part of the model. This makes them difficult to access from the script file and unreliable should we extend the conversion to look at other views. Therefore we decided to include the invariants in the documentation text box

---

[22] Microsoft Word97 is a trademark of the Microsoft Corporation

of the class' specification. The invariants are split into 2 kinds. Instance invariants are properties that hold between the attributes and relationships of a particular instance of the class. In keeping with the implicit self reference style of UML, we chose to allow the reference to a particular instance to be omitted. U2B will need to add the universal quantification over all instances of self. The last invariant is a class wide invariant that specifies properties that hold between the different instances of the class. Here, the quantification is an integral part of the property and must be given explicitly. Hence, U2B will not need to add quantification over instance references. The invariants are separated from any natural language description by the phrases INSTANCE INVARIANT: and CLASS INVARIANT: respectively. UML does not impose any particular notation for these operation and invariant constraint definitions, they could be described in natural language or using UML's associated Object Constraint Language (OCL). However some problems have been raised with OCL [9] and since we wish to end up with a B specification it makes sense to use bits of B notation to specify these constraints. The B has to observe a few conventions in order for it to become valid B within the context of the machine produced by U2B. For example, all operation outputs are called 'result'.

Example Translation

The example in Fig. 1 shows a class GAME that has typed and initialised attributes, parameterised operations (some with return values) and an aggregate relationship with another class, PRIZE. The class also uses another class, TICKET, as a type. Note the use of POW in some of the attribute types this is equivalent to an attribute with [0..n] multiplicity. The aggregate has a role name Prizes which will be used in its translation to B and a multiplicity which will effect its initialisation in the instance creation operation of GAME. Alongside the class diagram is shown the Rational Rose specification for the class GAME. Following the natural language description in the 'Documentation' box some instance and class invariants are given. The Atelier B proof tools were used to prove that these invariants were preserved by the operations of the example. In fact, it was found that the example contains an error and the class invariant, which says that a ticket must not belong to more than one game, is not upheld by the buy operation. Atelier B generated 24 proof obligations, 20 of which it automatically proved. The remaining 4 proof obligations required interaction in the form of direction on what hypotheses to use for the proof.

Each operation of the class also has a Rose Specification window with appropriate tabs for the definition of the operation. The operation pre-conditions and body are taken from the precondition and semantics tabs of the specification for the 'buy' operation in class GAME. The ANY construct is a statement of the B language that selects a value for a variable (here tt) satisfying some condition. In this case the condition is 'tt: TICKET - Tickets', i.e. select an unused ticket.

The precondition Tickets /= TICKET is not sufficient since it only ensures that there are tickets not used by the current game, whereas it should state that there are tickets not used by any game. Similarly the 'WHERE' restriction on tt should ensure that tt does not belong to any game whereas it only requires that tt is not in use by the current game. We are currently working on modifications to enable U2B to translate references to other instances of the class within operation specifications.

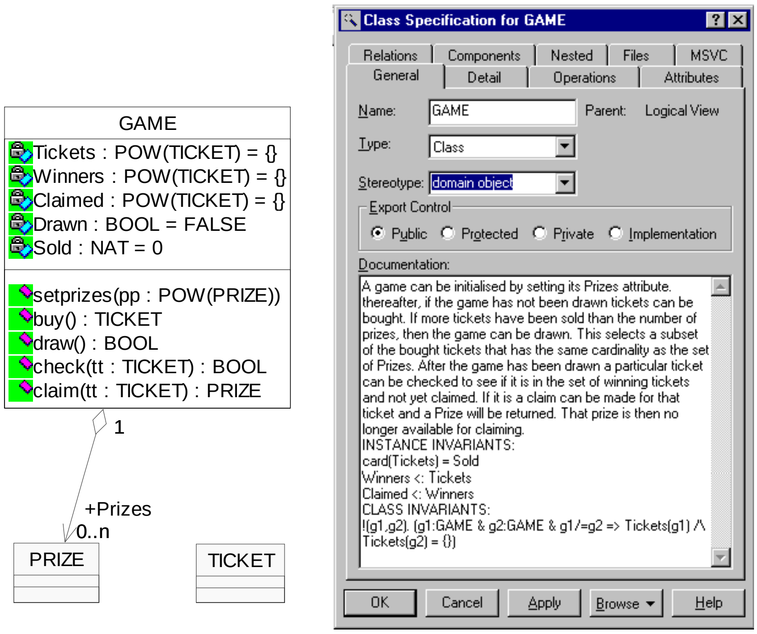

Fig. 1. Example Class Diagram and Class Specification

precondition

Prizes /= {}
Winners = {}
Tickets /= TICKET

semantics

ANY tt WHERE tt: TICKET - Tickets
THEN
Tickets := Tickets \/ {tt} ||
Sold := Sold +1 ||
result := tt
END

Below is shown the B Machine produced by U2B for the GAME class. The italicised text to the right of each line has been added as commentary.

MACHINE GAME
/*" A game can be initialised by setting its Prizes attribute. thereafter, if the game has…etc. "*/
SEES Bool_TYPE
USES PRIZE, — *U2B has detected the use of these other classes as*
TICKET — *types and provided references to them.*
SETS GAMESET — *This set represents all possible instances of the class.*
VARIABLES
GAMEinstances — *This variable is the set of current instances of the class.*
, Tickets, Winners, Claimed, Drawn, Sold, Prizes
A variable is declared for each attribute and for each aggregate.
INVARIANT — *Invariants define the type of each variable.*
GAMEinstances <: GAMESET

```
  & Tickets : GAMEinstances --> POW(TICKETinstances)              Attribute types are total
  & Winners : GAMEinstances --> POW(TICKETinstances)           functions from instances
  & Claimed : GAMEinstances --> POW(TICKETinstances)             to the type. The type is.
  & Drawn : GAMEinstances --> BOOL                                   derived from the class
  & Sold : GAMEinstances --> NAT                                                specification
  & Prizes : GAMEinstances --> POW(PRIZEinstances)       Aggregate types derived from other
                                                                  class and multiplicity of role.
  & !(g1,g2). (g1,g2:GAMEinstances & g1/=g2 => Tickets(g1) /\ Tickets(g2) = {})   Class invariant
  & !(self).(self:GAMEinstances => card(Tickets(self)) = Sold(self))   Instance invariants are
  & !(self).(self:GAMEinstances => Winners(self) <: Tickets(self))      included after adding the
  & !(self).(self:GAMEinstances => Claimed(self) <: Winners(self))   universal quantification !(self).
INITIALISATION
  GAMEinstances := {}                                     Initially the class has no instances
  || Tickets := {}      || Winners := {}                       …and therefore, no attributes.
  || Claimed := {}    || Drawn := {}
  || Sold := {}         || Prizes := {}

OPERATIONS
result  <-- GAMEcreate =                          A create operation is provided for each class.
PRE
  GAMEinstances  /= GAMESET.
THEN
  ANY new
  WHERE
    new : GAMESET - GAMEinstances                                      Pick any unused instance.
  THEN
    GAMEinstances := GAMEinstances \/ {new }                    Add it to the current instances.
    || Tickets := Tickets <+ {new |-> {}}                         Add a maplet from instance to the
    || Winners := Winners <+ {new |-> {}}                   initialisation value for each attribute
    || Claimed := Claimed <+ {new |-> {}}                                                      function.
    || Drawn := Drawn <+ {new |-> FALSE}
    || Sold := Sold <+ {new |-> 0}
    || Prizes := Prizes <+ {new |-> {}}                        Similarly for aggregate relationships.
    || result := new                                                           Return the new instance.
  END
END
;
result <--  buy (self) =                                Self added to parameter list automatically.
PRE
   self  :  GAMEinstances                                 Define type of self and other parameters
  & Prizes(self) /= {}                                    Additional pre-conditions from operation
  & Winners(self) = {}                                               specification (note addition of
  & Tickets(self) /= TICKETinstances                                  instance reference 'self').
THEN
  ANY tt WHERE tt: TICKETinstances - Tickets(self)
  THEN                                                                    Operation body from operation
      Tickets(self) := Tickets(self) \/ {tt} ||                                              specification
      Sold(self) := Sold(self) +1 ||
      result := tt
  END
END
;                                               other operations have been omitted for this illustration
```

Conclusions

We have implemented a first version of a practical tool that translates Rational Rose UML class diagrams to B. The tool is not complete; it requires extension to cope with other forms of association and generalisation. However such extension is technically feasible. The tool requires no intervention provided that constraint information is specified appropriately in the B notation. These constraints are one means of describing dynamic behaviour of operations in UML. Similarly, the B notation has been used to specify constraints, in the form of invariants, on the valid states of the system. We have translated an example UML class diagram and used the B Toolkit animator to explore its dynamic behaviour. We have attempted to use the Atelier B proof tools to verify that the dynamic behaviour of the examples operations preserves the invariants and in the process discovered an error in the example and an area that needs strengthening in the translator tool

References

1. J.R.Abrial The B Book - Assigning Programs to Meanings. Cambridge University Press, 1996 ISBN 0-521-49619-5
2. M.Buchi & R.Back. Compositional Symmetric Sharing in B. FM99 Vol.1 LNCS1708 pp431-451,1999
3. P.Facon, R.Laleau, & H.Nguyen. Mapping Object Diagrams into B Specifications. In Methods Integration Workshop, Electronic Workshops in Computing (eWiC), Springer Verlag. March1996
4. E.Meyer & J.Souquieres. A Systematic approach to Transform OMT Diagrams to a B specification. FM'99 Vol.1 LNCS 1708, pp875-895. Oct 1999
5. E.Meyer & T.Santen. Behavioural Conformance Verification in an Integrated Approach Using UML and B. In IFM'2000 : 2nd International Workshop on Integrated Formal Methods. Nov. 2000
6. N.Nagui-Raiss. A Formal Software Specification Tool Using the Entity-Relationship Model. In 13th International Conference on the Entity-Relationship Approach, LNCS 881. Dec.1994.
7. J.Rumbaugh, I.Jacobson & G.Booch. The Unified Modelling Language Reference Manual. Addison-Wesley, 1998. ISBN 0-201-30998-X
8. R.Shore. An Object-Oriented Approach to B. In Putting into Practice Methods and Tools for Information System Design - 1st Conference on the B method. Nov.1996
9. M.Vaziri & D.Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. Response to Object Management Group's Request for Information on UML 2.0, Dec. 1999
10. J.Warmer & A.Kleppe. The Object Constraint Language - Precise Modeling with UML. Addison-Wesley, 1999 ISBN 0-201-37940-6