Tool-Supported Use of UML for Constructing B Specifications

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 enable the behaviour of a system to be precisely specified and verified. However, even experienced users admit that creating useful models is difficult and this is a barrier to more widespread use. One reason for this is the lack of tools to assist in the modelling process. The process of formal specification is, in many ways, similar to that of programming where design notations and tools have evolved over many years. In this paper we suggest the adaptation of a graphical design notation (UML) for formal specification. In particular, we use class diagrams and statecharts from UML to construct B specifications. The process is supported with a prototype tool to perform automatic translation of UML into B specifications.

1. Introduction

In a survey we performed as part of previous work, experienced formal methods users said that reading and understanding formal specifications is not a significant problem [9]. With suitable training, programmers find formal specifications no more difficult to understand than programs. On the other hand, the same users reported that creating formal specifications is very difficult. They said that the main difficulty lies in finding useful abstractions. This indicates that the problem lies not in specifying the detailed semantics but in the preliminary stage of choosing the objects that make up the model and their representation as data structures.

The process of formal specification (in a model-based specification language) is, in many ways, similar to that of programming. Both involve modelling entities in a system using a precise notation and express desired behaviour upon data structures modelling state. However, techniques for programming have evolved over several decades, driven by a strong impetus to use computers to solve bigger and more complicated problems. Formal specification has developed to some extent but not with such a strong motivation. Now that the problems being tackled are more complicated the need for formal specification is greater, but it lags behind programming in terms of tools and techniques to aid the modelling process. We believe that graphical modelling tools similar to those used for program design would aid the process of formal specification. With this in mind we have used diagrammatic notations of the UML [7] for formal specification. To support this we have developed a prototype tool to convert adapted forms of UML class diagrams and statechart diagrams into specifications in the B language [1]. The aim is to use some of the features of UML diagrams to make the process of writing formal specifications easier, or at least more approachable to average programmers. We view this work as a feasibility investigation rather than a final method or product. The translation relies on precise expression of additional behavioural constraints in the specification of class diagram components and in statecharts attached to the classes. These constraints are described in an adapted form of the B 'abstract machine notation'. The type of class diagrams that can be converted is restricted in order to comply with constraints of the B-method without making the B unnatural. The resulting UML model is a precise formal specification but in a form which is more friendly to average programmers, especially if they use the same UML notation for their program design work. The diagrammatic notation and tool support brings its benefits to the modelling process for formal specification. The translation to textual B specification does not add anything to the specification; it merely provides an alternative textual form. In this textual form, however, the benefits of the B method are obtained. The translation also demonstrates the validity of the graphical forms and defines their semantics. We envisage benefits to B users (especially novices) from being able to develop models in the UML diagrammatic form and we see this as a possible way to overcome some of the psychological barriers that programmers have against formal specification.

2. Benefits of a Diagrammatic Form for Specification

The majority of students on computer science courses express an aversion to formal specification whereas they are quite comfortable using graphical program design notations such as the UML. We believe that this is largely an unwarranted fear and that formal specification, given the same level of tool and language support should be no more difficult than programming. Advantages of graphical design aids are more to do with the creation of models than with conveying information. Graphical descriptions can be misleading to read, they often convey different meanings to different readers and require experience to interpret secondary features [6] but to the writer they provide a quick way to express their ideas and to assist in visualizing prototype models that must otherwise be built entirely within the mind. Textual representations, although often more accurate in conveying precise meanings, are much more cumbersome for creating some aspects of these models. Graphical representations are good for helping to visualize structures, composition and the relationships between elements. Modelling large systems usually requires initially a structural design, which is then populated with more precise semantic detail. It is this first modelling stage that benefits from program design tools such as UML. Class diagrams allow the types of objects in the problem domain and the relationships between them to be modelled, visualized, prototyped and altered quickly. Attempts to add the semantic detail to these models may result in deficiencies in the model being discovered and lead to refinements to the model. These changes can be made quickly because the model is highly visible and easily alterable with the aid of the graphical design tools. Readability and ambiguity is not an issue because it is the creators that are using the tools for modelling. These features have made graphical design techniques such as UML popular for developing programs. We contend that the process of writing formal specifications is in some ways similar to programming and involves similar difficulties in abstraction, look-ahead (predicting the subsequent use of these abstractions) and viscosity (difficulty of making structural changes). Although we do not present a cognitive dimensions analysis here, we note that these terms are taken from the framework proposed by Green [12] for comparison of information systems notations. Therefore tools that programmers have evolved for writing programs, or ones very similar to them, should bring similar benefits when writing formal specifications. In particular the UML and associated tools attack viscosity in order to alleviate the difficulty of choosing and committing to appropriate abstractions.

3. The B Language and Toolkit

The B language is a formal specification notation that has strong structuring mechanisms and good tool support. There are two 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 ('INCLUDES') another machine. If machine A includes machine B, the state of B is visible to A and alterable via B's operations. Another form of machine inclusion is ‘EXTENDS’. This is the same as INCLUDES but makes the included machines operations accessible as if they were operations of the including machine. 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 (or extended) by one other machine.

4. Benefits of Translating UML to B

A will be seen, the translatable UML model with formal annotations is just as precise and complete as the equivalent B specification. This is demonstrated by the fact that it can be translated to B automatically. However, there are still benefits to translating into a B specification:

- The textual B specification may be more readable to experienced formal methods users
- The B specification is mathematically manipulable enabling reasoning and proof to be performed
- The B toolkit is available for type analysis, proof assistance and animation
- The translation demonstrates the semantics of the UML version.

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. Animation is useful, especially to novices, because it provides feedback and debugging of the specification. It is also useful for validation, i.e. demonstrating to users that the specification describes a system which will be useful.

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 associations with other objects) that disobeys the invariant. A safety or business critical property of the system could be specified and verified in this way.

UML models prepared for translation to B contain invariant and method specifications written in B notation. The annotated UML diagram is given a precise semantics as expressed by its equivalent form in the B notation as generated by the translator.

5. U2B Class Diagram Translator

The U2B translator converts Rational Rose[1] UML Class diagrams, including attached statecharts, 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") in Rose. U2B uses the object-oriented libraries of the Rose Extensibility Interface to extract information about the classes in the logical diagram of the currently open model. The object model representation of the UML diagram means that information is easily retrieved and the program structure

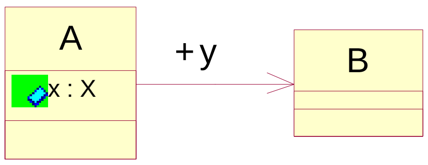
[1]Rational Rose is a trademark of the Rational Software Corporation

can be based around the logical information in the class rather than a particular textual format. U2B uses Microsoft Word[2] to generate the B Machine files.

5.1 Translation of Structure and Static Properties

The translation of Classes, attributes and operations is derived from proposals for converting OMT to B [3]. A separate machine is created for each class and this contains a set of all possible instances of the class and a variable that represents the subset of current instances of the class. Attributes and (unidirectional) associations are translated into variables whose type is defined as a function from the current instances to the attribute type (as defined in the Class diagram) or associated class. Attribute types may be any valid B expression that defines a set. This includes predefined types such as NAT and NAT1, boolean and string (if translating to B-Core B, the appropriate B library machines must be referenced via a SEES clause in the class' specification documentation window) functions, sequences, powersets, instances of another class (referenced by the class name) or enumerated or deferred sets defined in the class specification documentation window. If the type involves another class the machine for that class will either be extended (EXTENDS) by this machine if there is a path of unidirectional associations to that class or otherwise used (USES). Any references to the class in type definitions of variables or operation arguments will be changed to the current instances set for that class.

For example consider the following class diagram with classes *A* and *B*, where *A* has an attribute *x* and there is an association *y* from *A* to *B*:



This will result in the following machine representing all instances of *A*:

```
MACHINE  A
EXTENDS  B
SETS  ASET
VARIABLES  Ainstances, x, y
INVARIANT
     Ainstances <: ASET &
     x : Ainstances --> X &
     y : Ainstances --> Binstances
. . .
END
```

A separate machine will be generated for class *B*.

It is often useful to define types as enumerated or deferred sets for use in many machines. We use 'class utilities' for this. In UML, a class utility is a class that doesn't

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

have any instances, only static (class-wide) operations and attributes. The U2B translator creates a machine for each class utility and copies any text in the specification documentation window of its class specification into the machine. Hence definitions of sets and constants can be described in B clauses in the documentation window.

Such sets can also be defined locally to a class in the class's specification documentation window. Any valid B clause can be added in this window. For example, we use this method to specify invariants for the class. Each clause must be headed by its B clause name in capitals and starting at the beginning of a line, the text that follows that clause, up until the next clause title (if any) will be added to the appropriate clause in the machine. Any text before the first clause is treated as comment and added as such at the top of the machine

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. Note that, according to our definition (via translation) of class diagrams, association means that the source class is able to invoke the methods of the target class.

Often, a B machine models a single generic instance of an entity, rather than an explicit set of instances (in the same way that a class in UML leaves instance referencing implicit). The resulting specification is simpler and clearer for not modelling multiple instances. The U2B translator creates a single-instance machine if the class multiplicity (cardinality) is set to 1..1 in the UML class specification. Note that this can only be done at the top level of a structure since at lower levels the instance set is used for referencing by the higher level.

The B method imposes some restrictions on the way machines can be composed. These restrictions ensure compositionality of proof. Their impact is that no write sharing is allowed at machine level (i.e. a machine may only be included or extended by one other machine). Also, the inclusion mechanism of B is hierarchical so that, if *M1* includes *M2*, then *M2* cannot, directly or transitively, include *M1*. We reflect these restrictions in the UML form of the specification, which must therefore be tree like in terms of unidirectionally related classes. Non-navigable (and bi-directional) associations are ignored but may be used to illustrate the use of another class as a type (i.e. read access only). However, multiple, parallel associations between the same pair of classes are permitted.

Although we would like to adhere to the UML class diagram rules as much as possible, since our aim is to make B specification more approachable rather than to formalise the UML we are relatively happy to impose restrictions on the diagrams that can be drawn. That is, we only define translations for a subset of UML class diagrams. Other authors [2],[3],[4],[5],[8] have suggested ways of dealing with the translation of more general forms of class diagrams. However, the structures of B machines that result from these more general translations can be cumbersome. If the specification were written directly in B, it would be highly unlikely that the resulting B would have this form. Since we also desire a usable B specification we prefer to restrict the types of diagrams that can be drawn.

5.2 Dynamic Behaviour

The dynamic behaviour modelled on a class diagram that is converted to B by U2B is embodied in the behaviour specification of classes operations and in invariants specified for the classes. These details are specified either in a textual format as annotation to the class diagram or in a statechart attached to the class. An operation may be specified completely by textual annotation, completely by statechart transitions, or by a combination of both composed as simultaneous specification.

Textual behaviour specification - In Rational Rose, 'Specifications' are provided for operations (as well as many other elements) and these provide text boxes dedicated to writing pre-conditions and semantics for the operation. Unfortunately there is no text box for a class invariant. One suggestion is to put invariant constraints in a note attached to the class [11], but notes are 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 translation program and unreliable should we extend the conversion to look at other views. Therefore we include the invariants as a clause in the documentation text box of the class' specification window. The invariants are generally of two kinds, instance invariants (describing properties that hold between the attributes and relationships within a single instance) and class invariants (describing properties that hold between different instances). For instance invariants, in keeping with the implicit self-reference style of UML, we chose to allow the explicit reference to this instance to be omitted. U2B will add the universal quantification over all instances of the class automatically. For class invariants, the quantification over instances is an integral part of the property and must be given explicitly. Hence, U2B will not need to add instance references.

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 Object Constraint Language (OCL). However some problems have been raised with OCL [10] 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 constraints are specified in a notation that is close to B notation but has to observe a few conventions in order for it to become valid B within the context of the machine produced by U2B. When writing these bits of B the specifier shouldn't need to consider how the translation would represent the features (associations, attributes and operations) of the classes. Also we felt we should follow the more object-oriented conventions of implicit self referencing and use of the dot notation for explicit instance references. Therefore, when writing the constraints, a dot notation is used to reference the ownership of features. For example, if class *A* has an association to class *B*, we might write *AassocB.Battr*, where *AassocB* is an association from class *A* to class *B* and *Battr* is an attribute of class *B*. This would be translated to *Battr*(*AassocB*(*thisA*)).

UML operation signatures contain a provision for specifying the type for a value returned by the operation. Since *B* infers this from the body of the operation we use it instead to name the identifiers that represent operation return values.

Statechart Behavioural Specification - For classes that have a strong concept of state change, a statechart representation of behaviour is appropriate. In UML a statechart model can be attached to a class to describe its behaviour. A statechart model consists of a set of states and transitions that represent the state changes that are allowed. If a statechart model is attached to a class the U2B translator combines the behaviour it describes with any operation semantics described in the operation specification semantics windows. Hence operation behaviour can be defined either in the operation semantics window or in a statechart model for the class or in a combination of both.

The name of the statechart model is used to define a state variable. (Note that this is not the name of a statechart diagram, several diagrams could be used to draw the statechart model of a class). The collection of states in the statechart model is used to define an enumerated set that is used in the type invariant of the state variable. The state variable is equivalent to an attribute of the class and may be referenced elsewhere in the class and by other classes. Statechart transitions define which operation call causes the state variable to change from the source state to the target state, i.e., an operation is only allowed when the state variable equals a state from which there is a transition associated with that operation. To associate a transition with an operation, the transition's name must be given the same name as the operation. Additional guard conditions can be attached to a transition to further constrain when it can take place. All transitions cause the implicit action of changing the state variable from the source state to the target state. (The source and target state may be the same). Additional actions can also be attached to transitions. The translator finds all transitions associated with an operation and compiles a SELECT substitution of the following form and composes it with the operation body from the textual specification from the operations semantics window:

```
SELECT statevar=sourcestate1 & sourcestate1_guards
THEN statevar:=targetstate1 || targetstate1_actions
WHEN statevar=sourcestate2 & sourcestate2_guards
THEN statevar:=targetstate2 || targetstate2_actions
etc
END ||
<operation body from semantics window>
```

Hence the transition guards constrain the conditions under which an operation can be invoked. If none of the guards in the SELECT are true then the operation cannot proceed at all, not even the operation body from the textual specification[3]. One effect of this is that the order that operations can be invoked is constrained thus reflecting the statechart diagram. Actions should be specified on the transitions when the action differs depending on the state transition. Where the action is the same for all the transitions relevant to that operation it should be specified in the operation semantics window.

[3] Provided the operation body has no precondition.

6. Example

The example in Fig. 1 is a model of a railway station. It is an extension of the example in [13]. The class diagram consists of a class *STATION* that has a typed and initialised attribute, parameterised operations (one with a return value), and an association with another class *PLATFORM*. The association has a role name *platforms*, which is used to refer to instances of the associated class.

In Fig.2 the Rational Rose specification window for the class *GAME* is shown. Following the natural language description in the documentation box some class invariants are given. The first part of the invariant contains three instance invariants that implicitly apply to all platform instances. The final part of the invariant is a class invariant that explicitly quantified for all pairs of platforms. Note that some of these invariants refer to a state variable, *platfom_state*, and its possible values that have been obtained from a state chart attached to the class.

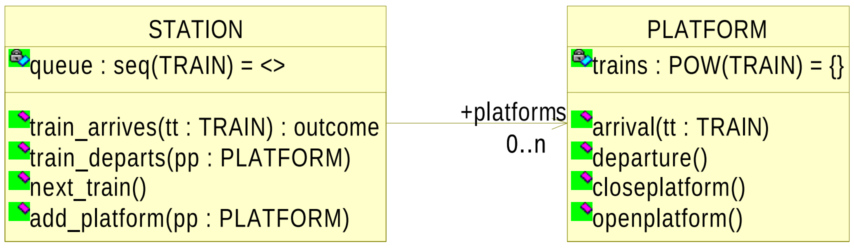
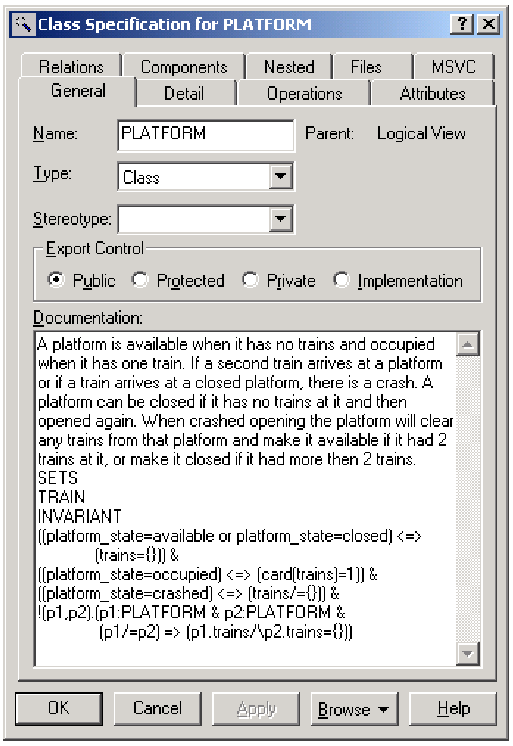


Fig. 1. Class Diagram

Each operation of the class also has a Rose Specification window with appropriate tabs for the definition of the operation. The operation pre-condition and body, shown in Fig.3, are taken from the precondition and semantics tabs of the specification for the *train_arrives* operation in class *STATION*. The precondition states that *tt* must not belong to the range of queue and it must not belong to the union of the set *trains* for all platforms associated with this station. That is, the arriving train must not be waiting to get into the station or at a platform already. If an empty platform exists at this station, the operation sends the train to any such empty platform and returns the outcome *in_station*. Note that the arrival at a platform is handled by calling the *arrival* operation of class *PLATFORM*, specifying the selected platform, *pp*, using the dot prefix notation. If no platform is available the train is appended to the queue and an outcome, *waiting*, is returned.



Name: PLATFORM Parent: Logical View

Type: Class

Stereotype:

Export Control: [✓ Public] Protected Private Implementation

Documentation:

```
A platform is available when it has no trains and occupied
when it has one train. If a second train arrives at a platform
or if a train arrives at a closed platform, there is a crash. A
platform can be closed if it has no trains at it and then
opened again. When crashed opening the platform will clear
any trains from that platform and make it available if it had 2
trains at it, or make it closed if it had more then 2 trains.
SETS
TRAIN
INVARIANT
((platform_state=available or platform_state=closed) <=>
          (trains={})) &
((platform_state=occupied) <=> (card(trains)=1)) &
((platform_state=crashed) <=> (trains/={})) &
!(p1,p2).(p1:PLATFORM & p2:PLATFORM &
          (p1/=p2) => (p1.trains/\p2.trains={}))
```

Fig. 2. Class specification window for the class PLATFORM showing natural language description and invariants

train_arrives precondition

```
tt/: ran(queue) &
tt/: UNION(pp).(pp:platforms|pp.trains)
```

train_arrives semantics

```
IF #(qq).(qq:platforms & qq.platform_state=available)
THEN
    ANY pp WHERE
         pp:platforms &
         pp.platform_state=available
    THEN
         pp.arrival(tt) ||
         outcome:=in_station
    END
ELSE
    queue:=queue^[tt] ||
    outcome:=waiting
END
```

Fig. 3. Precondition and semantics for operation train_arrives of class Station

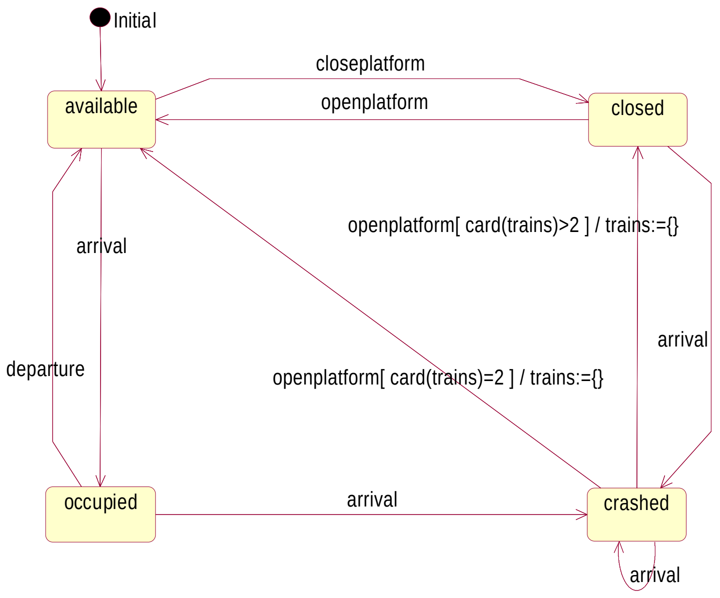


Fig. 4. State chart attached to class platform

Fig. 4 shows the state chart attached to class platform. The state chart describes the states that a platform can be in and which transitions between states are possible. Each transition corresponds to an operation of the machine and has its event named after an operation. For example, when a platform is in the state *available*, two operations are allowed: *closeplatform* and *arrival*. Execution of the *arrival* operation in this state changes the control state to *occupied*, while executing the *closeplatform* operation changes the control state to *closed*. The transitions associated with the operation *openplatform* have additional guards which determine which of the transitions will be taken when openplatform occurs from the state crashed. These transitions also take a different action from the *openplatform* transition that occurs from the state closed.

Below is shown the B machine for the class *PLATFORM*. (Note that the layout has been altered to fit the format and space constraints of this paper).

```
MACHINE PLATFORM
/*" A platform is... etc. "*/

SETS
  PLATFORMSET;
  PLATFORM_STATE={available,closed,occupied,crashed} ;
  TRAIN
```

PLATFORMSET is the set of all possible instances of *PLATFORM*. *PLATFORM_STATE* has been generated from the states on the attached statechart.

TRAIN has been generated from the SETS machine clause in the class specification documentation window.

```
VARIABLES
   PLATFORMinstances,
   platform_state,
   trains
INVARIANT
   PLATFORMinstances  <: PLATFORMSET &
   platform_state : PLATFORMinstances --> PLATFORM_STATE
&
   trains : PLATFORMinstances --> POW(TRAIN) &
```

PLATFORMinstances is a variable subset of *PLATFORMSET*, representing the current instances of *PLATFORM*. A variable, *platform_state* represents the state that each *PLATFORMinstance* is in. A variable, *trains*, represents the subset of TRAIN belonging to each *PLATFORMinstance*.

```
!(thisPlatform).(thisPlatform:PLATFORMinstances =>
 ((platform_state(thisPlatform)=available or
   platform_state(thisPlatform)=closed) <=>
               (trains(thisPlatform)={})) &
 ((platform_state(thisPlatform)=occupied) <=>
               (card(trains(thisPlatform))=1)) &
 ((platform_state(thisPlatform)=crashed) <=>
               (trains(thisPlatform)/={})) &
 !(p1,p2).(p1:PLATFORMinstances & p2:PLATFORMinstances
&
               (p1/=p2) => (trains(p1)/\trains(p2)={}))
)
```

Further invariants reflect the invariants specified in the specification documentation text box for class *PLATFORM* (see Fig.2). Universal quantification over *PLATFORMinstances* has been added and the dot notation of explicit instance references has been converted into parameters.

```
INITIALISATION
   PLATFORMinstances := {} ||
   platform_state := {} || trains := {}
OPERATIONS
 Return  <-- PLATFORMcreate =
  PRE  PLATFORMinstances  /= PLATFORMSET
  THEN
   ANY new
   WHERE
     new : PLATFORMSET - PLATFORMinstances
   THEN
     PLATFORMinstances := PLATFORMinstances \/ {new} ||
     platform_state(new):=available ||
     trains(new):={} ||
     Return := new
   END
  END ;
```

Initially *PLATFORMinstances* is empty, and hence all variables are empty sets. A create operation is provided which non-deterministically picks any unused instance from *PLATFORMSET* and initialises state and attribute variables to the initial values given in the statechart and UML class specification respectively

```
  arrival (thisPlatform,tt) =
   PRE
     thisPlatform  :  PLATFORMinstances &
     tt:TRAIN &
     tt/:UNION(pp).(pp:PLATFORMinstances|trains(pp))
   THEN
     SELECT platform_state(thisPlatform)=available
     THEN   platform_state(thisPlatform):=occupied
     WHEN   platform_state(thisPlatform)=closed
     THEN   platform_state(thisPlatform):=crashed
     WHEN   platform_state(thisPlatform)=occupied
     THEN   platform_state(thisPlatform):=crashed
     WHEN   platform_state(thisPlatform)=crashed
     THEN   skip
     END ||
     trains(thisPlatform):=trains(thisPlatform) \/ {tt}
   END ;

   openplatform (thisPlatform) =
   PRE
     thisPlatform  :  PLATFORMinstances
   THEN
     SELECT platform_state(thisPlatform)=closed
     THEN   platform_state(thisPlatform):=available
     WHEN   platform_state(thisPlatform)=crashed &
            card(trains(thisPlatform))=2
     THEN   platform_state(thisPlatform):=available ||
            trains(thisPlatform):={}
     WHEN   platform_state(thisPlatform)=crashed &
            card(trains(thisPlatform))>2
     THEN   platform_state(thisPlatform):=closed ||
            trains(thisPlatform):={}
     END
   END
  END
```

Operations are defined for each operation of the class. (Only the two operations are shown). A parameter, *thisPlatform*, has been added to define the instance that the operation is to operate on; this is implicit in the UML class diagram version. The type of this and any other parameters are defined as operation preconditions. Other preconditions are derived from the operation pre-conditions specification window of the class diagram. The operation body is derived from the operation semantics specification window of the class diagram (see Fig 3) and from the statechart diagram. The body of operation *arrival* consists of a 'SELECT' guard, which defines the state transitions that take place when this operation (event) occurs, and, in parallel, the action specified in the semantics window, which occurs irrespective of state transition. In operation *openplatform* additional conditions determine the final state when the initial state is *crashed* leading to two different SELECT branches for the *crashed* state.

The B machine for the class *STATION* does not model instances and therefore variables representing attributes and associations are typed directly rather than as functions. This machine EXTENDS the *PLATFORM* machine so that it can call operations of *PLATFORM* if required. (EXTENDS is used so that multiple layers of associations can be navigated)

```
MACHINE STATION
/*" A station can have several platforms…etc. "*/

EXTENDS
   PLATFORM
SETS
   MSG={in_station,waiting}
VARIABLES
   queue,
   platforms
INVARIANT
   queue : seq(TRAIN) &
   platforms : POW(PLATFORMinstances) &
   ran(queue)/\UNION(pp).(pp:platforms|trains(pp)) = {}
&
   size(queue)=card(ran(queue))
INITIALISATION
   queue:=<> ||
   platforms := {}
```

A variable *platforms*, which is a subset of *PLATFORMinstances*, is used to model the association with class *PLATFORM*. No *create* operation is generated because instances are not modelled. Instead, the variables are initialised in the INITIALISATION clause to the values specified in the class diagram (in this case both are initialised to empty). The precondition and semantics for the *train_arrives* operation of *STATION* shown in Fig. 3 is as follows:

```
outcome <--  train_arrives (tt) =
 PRE
   tt:TRAIN &
   tt/: ran(queue) &
   tt/: UNION(pp).(pp:platform|trains(pp))
 THEN
   IF #(qq).(qq:platforms &
            platform_state(qq)=available)
   THEN
       ANY pp WHERE
            pp:platforms &
            platform_state(pp)=available
       THEN
            arrival(pp,tt)  ||  outcome:=in_station
       END
   ELSE
       queue:=queue^[tt]  ||  outcome:=waiting
   END
 END ;
```

This operation makes various references to components of the *PLATFORM* class, including reading the state variable *platform_state* and calling its *arrival* operation. The object oriented dot notation of Fig. 3 has been changed to standard B notation.

7. Further Work

The translator could be improved in several areas. We intend to extend it to deal with various options for the creation of class instances that are implied by different types of relationship (such as composition). Other extensions to its functionality could be made, for example to handle inheritance relationships or to perform checking of the diagram before attempting translation, but these are beyond the scope of our current aims.

We intend to evaluate the claim that the diagrammatic formal specification is beneficial during creation of a formal specification by writing a real life example in B-UML and in B and comparing the two.

8. Conclusions

A graphical modelling tool is invaluable for developing structural models of systems. This has led to the popularity of tool supported modelling languages such as UML. By adding precise semantic details in the form of specification texts and defining a particular meaning to the diagrammatic features we can interpret some UML diagrams as formal specifications. We have implemented a prototype add-in tool that translates these diagrammatic specifications to B. We believe that the diagrammatic form of formal specification will assist in the difficult task of creating appropriate models and will make formal specification more approachable (especially to novices). Tool support, such as that provided by U2B is essential for making this practical.

Acknowledgements

This work was funded in part by the EU Research Project MATISSE (IST-1999-11435) and in part by EPRSRC as a postgraduate research project.

References

1. Abrial, J.R. (1996) The B Book - Assigning Programs to Meanings. *Cambridge University Press*, ISBN 0-521-49619-5

2. Facon, P., Laleau, R., & Nguyen, H. (1996) Mapping Object Diagrams into B Specifications. *In Methods Integration Workshop, Electronic Workshops in Computing (eWiC),* Springer Verlag.
3. Meyer, E. & Souquieres, J. (1999) A Systematic approach to Transform OMT Diagrams to a B specification. *FM'99* LNCS1708 1, 875-895
4. Meyer, E. & Santen, T. (2000) Behavioural Conformance Verification in an Integrated Approach Using UML and B. *In IFM'2000 : 2nd International Workshop on Integrated Formal Methods.*
5. Nagui-Raiss, N. (1994) A Formal Software Specification Tool Using the Entity-Relationship Model. *In 13th International Conference on the Entity-Relationship Approach*, LNCS 881.
6. Petre, M. (1995) Why Looking Isn't Always Seeing. *Comm. Of the ACM*. 38(6)
7. Rumbaugh, J., Jacobson, I. & Booch, G. (1998) The Unified Modelling Language Reference Manual. *Addison-Wesley*, . ISBN 0-201-30998-X
8. Shore, R. (1996) An Object-Oriented Approach to B. *In Putting into Practice Methods and Tools for Information System Design - 1st Conference on the B method*.
9. Snook, C. and Harrison, R. (2001) Practitioners Views on the Use of Formal Methods: An Industrial Survey by Structured Interview. *To be published in Information and Software Technology*
10. Vaziri, M. & Jackson, D. (1999) Some Shortcomings of OCL, the Object Constraint Language of UML. *Response to Object Management Group's Request for Information on UML 2.0*
11. Warmer, J. & Kleppe, A. (1999) The Object Constraint Language - Precise Modeling with UML. *Addison-Wesley,* ISBN 0-201-37940-6
12. Green, T.R.G (1989) Cognitive Dimensions of Notations. In A.Sutcliffe and L.Macaulay (Eds.) People and Computers V. Cambridge: Cambridge University Press.
13. Lano, K. (1996) *The B Language and Method*. Springer.