UML-B: Formal Modeling and Design Aided by UML

COLIN SNOOK and MICHAEL BUTLER
University of Southampton

The emergence of the UML as a de facto standard for object-oriented modeling has been mirrored by the success of the B method as a practically useful formal modeling technique. The two notations have much to offer each other. The UML provides an accessible visualization of models facilitating communication of ideas but lacks formal precise semantics. B, on the other hand, has the precision to support animation and rigorous verification but requires significant effort in training to overcome the mathematical barrier that many practitioners perceive. We utilize a derivation of the B notation as an action and constraint language for the UML and define the semantics of UML entities via a translation into B. Through the UML-B profile we provide specializations of UML entities to support model refinement. The result is a formally precise variant of UML that can be used for refinement based, object-oriented behavioral modeling. The design of UML-B has been guided by industrial applications.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Design, Verification

Additional Key Words and Phrases: Modeling, refinement, UML-B

1. INTRODUCTION

Formal specifications provide a precise supplement to natural language descriptions and can be rigorously validated and verified, leading to the early detection of errors. Although academic interest in formal methods has been lively, with many research groups throughout the world, uptake within industry has been limited. Amey [2004] reports that customers are often "aghast" at the idea of formal methods being used to develop their products and suggests UML instead. He suggests "formality by stealth" and cites semantically strengthened UML as an example. However, we shouldn't impose formal methods on practitioners against their will [Glass 2004], but aim to address some of the barriers

Authors' address: Electronics and Computer Science, University of Southampton, SO17 1BJ, UK; email: {cfs,mjb}@ecs.soton.ac.uk.

practitioners face. In a survey of industry, Craigen et al. [1995] concluded, "Successful integration is important to the long term success of formal methods." An advantage of a formal method based on the UML is that it minimizes the cost and risk of adoption by integrating with existing methods.

1.1 B

The B language [Abrial 1996] is a state model-based, formal specification notation, designed to support formally verified development by refinement from specification through to implementation. Formal verification of proof obligations ensures that a specification is consistent throughout its refinements. B, like its predecessor, Z [Spivey 1988], is based on set theory and first-order predicate logic. For refinement, B requires a 'refinement relation' as part of its invariant predicate, which is analogous to an 'abstraction relation' schema in Z. An invariant in B is a property of the state that operations are expected to maintain. Although B provides facilities for generation of executable code, it can also be used as an abstract specification language similar to Z. B benefits from commercial tool support for proof, Atelier-B [ClearSy 2003] and the B-Toolkit [B-Core 1996]. However, proof is difficult for inexperienced practitioners. A more accessible, automated form of verification is model checking [Clarke et al. 1999]. A B model checker, ProB, has been developed at the University of Southampton [Leuschel and Butler 2003]. It is equally important to ensure that the specification is a useful one. ProB includes an animator, which allows us to observe and validate the simulated behaviour of a specification

A B module consists of a number of B components from the most abstract specification, through possibly many refinements. B provides a structuring mechanism (includes) to decompose a component so that parts of the state can be encapsulated and segregated, making them easier to comprehend, reason about and manipulate. If a component, C, includes a machine M, the state of M is visible to C but only alterable via M's operations. Since machine inclusion is intended to provide independently provable units, shared write access is disallowed.

The example in Figure 1 is a telephone book expressed as a B machine. Invariants are used to define the type of each variable. In this case, the variable, NAME, represents the set of names that are currently in the phone book. NAME is declared as belonging to the powerset[1] of NAME_SET, the set of all possible names. The variable, pbook, represents the phone book mapping names to numbers. pbook is declared to be an injective[2] function ensuring that numbers in the phonebook are unique to names. Initially, pbook and NAME are both empty. In the machine's operations, preconditions define the type of any arguments. Additional 'guards' may be specified on the arguments or on state variables. For example, in the add operation, numb must not belong to the range of pbook. Also in the add operation an unused name is selected non-deterministically using an ANY selector and its corresponding phonebook

[1]The powerset, $\mathbb{P}(S)$ of a set, S, is the set of all subsets of S.
[2]An injective function is one in which each element of the range is mapped to by at most one element of the domain.

```
MACHINE          phonebook
SETS             NUMB ; NAME_SET
VARIABLES        NAME , pbook
INVARIANT        NAME ∈ ℙ(NAME_SET) ∧ pbook ∈ NAME ⤖ NUMB
INITIALISATION   NAME := ∅ || pbook := ∅
OPERATIONS
  add (numb) =
    PRE numb ∈ NUMB THEN
      SELECT numb ∉ ran(pbook) THEN
        ANY name WHERE name ∈ NAME_SET - NAME THEN
          NAME := NAME ∪ {name} || pbook(name):=numb
        END
      END
    END;
  remove (name) =
    PRE name ∈ NAME THEN
      NAME := NAME - {name} ||  pbook := {name} ⩤ pbook
    END;
  numb ← lookup (name) =
    PRE name ∈ NAME THEN
      numb := pbook(name)
    END
END
```

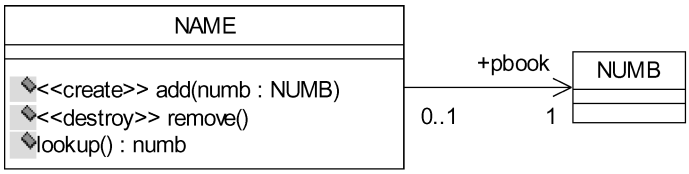
Fig. 1. B specification of a telephone book.

number is set, pbook(name):=numb, via indexed assignment. Operation behaviour is defined via 'substitutions' (equivalent to postconditions) that show how the final state of machine variables depends on their initial state and the arguments. Operations may return values as defined at the beginning of the operation signature (e.g. numb in operation lookup). Other symbols used in the example are: set union, ∪, and domain subtraction.[3]

1.2 Formality and the UML

The Unified Modelling Language (UML) [Rumbaugh et al. 1998] has been criticized for lacking a formal semantics. However, this may have contributed to the growth of the UML by allowing experimentation. A key feature of UML is its extensibility mechanisms, which allow users to develop their own semantic profiles for particular modelling domains. Although a stronger semantic underpinning is provided for UML 2.0, it is not clear that this will result in a notation and tool support suitable for formal proof of refinement. The UML includes a formal constraint notation, OCL [Warmer and Kleppe 2003], which aims to be approachable to practitioners by avoiding mathematical symbols. OCL has been criticized by some formal methods users for being cumbersome compared to traditional set based modelling notations [Vaziri and Jackson 1999]. Tool support for OCL is lacking compared to B although this is improving [Toval et al. 2003].

[3]Domain subtraction removes all the maplets of a relation that emanate from the elements in the given set.



operation	guard	action
add	numb∉ran($pbook)	pbook:=numb
remove	-	-
lookup	-	numb:=pbook

Fig. 2. UML-B model of a telephone book.

This article introduces a profile of the UML called UML-B [Snook et al. 2004], illustrates its application through some small case studies, and outlines how formal refinement may be applied to UML-B models. UML-B is precise and semantically well defined via equivalence to B and includes a constraint and action language, μB, which is derived from B. A translator tool, U2B [Snook and Butler, 2004], is available so that B verification and validation tools can be used. To give a flavour of UML-B, consider the specification of the telephone book in Figure 2. 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 table (Figure 2) shows μB conditions and actions for some of the operations. The add operation of class NAME has the stereotype ≪create≫, which means that it adds a new name to the class. It takes a parameter numb, which must be an instance of the class, NUMB, but not already used in a link of the association pbook (see μB operation guard), and uses this as the link for the new instance (see μB operation action). The remove operation has no μB action; its only action is the implicit removal of this from the class NAME. This specification is equivalent to the B version introduced in Figure 1 and applying the U2B tool to this UML-B model results in the B model of Figure 1.[4]

1.3 Influence of Industrial Projects on the Development of UML-B

Our initial U2B translation used concepts of modelling class instances proposed by other authors such as Meyer and Souquières [1999]. We added behavioural modelling by state machines with similarities to Sekerinski's work [1998]. During the MATISSE project [MATISSE 2003] we developed additional features to better support state machine modelling and refinement of state machine models. We also added class instance modelling features that suit embedded systems (which often have a small number of pre-existing instances). During the PUSSEE project [Mermet 2004] we found our original approach to combining UML and B to be restrictive and adopted an alternative modelling style where an entire package is translated into a single B component. To support this style, we defined different package stereotypes for refinement and

[4]The automatically generated B will use the B macro facility, which has been suppressed in Figure 1.

to support model decomposition. We also developed additional modelling features and alternative translation strategies so that the most appropriate mode can be adopted to support verification proof. UML-B was evaluated by a safety-critical software specialist on a case study involving complex data modelling and as a result several new features for instance modelling with class specialization were introduced. These issues will be developed further in the remainder of this article.

1.4 Difficulties Translating from UML to B

B has many features similar to UML, such as encapsulation of operations with associated state variables. However a simple translation from classes to machines is problematic and other mappings are needed. This section identifies features of the B language that make it difficult to map object-oriented models to B. These features are, in general, due to the main purpose of B, which is to facilitate modular proof of large systems. The main motivation for translating UML into B is to enable design refinements to be formally proven. Therefore, for a translation to be useful, it is important that the B is reasonably natural and does not complicate the proof process.

B is not object-oriented. A fundamental feature of object-orientated methods is the ability to model classes of objects via abstract data types. B has an encapsulation mechanism (machines) that allows variables to be grouped with the operations that act upon them. It is possible, via machine renaming, to instantiate several enumerated instances of a machine. However, there is no mechanism to 'lift' a specification to an indeterminate or variable set of instances. For example, Z has 'promotion,' which enables schemas to be used to define a behaviour that is then promoted and bound to a set of instances at a higher level. We overcome this limitation by explicitly modelling the set of instances and modelling each class feature with a function from the set of instances as will be described later.

Restrictions on B component and variable access. B restricts the way that operations can be called between and within machines. These restrictions are necessary in order to achieve composition of proof.

1. A machine cannot have more than one other machine that makes calls to its operations. This means that if a class-machine mapping is used, only one other class can access a class.
2. There must not be any loops within the calling structure of a set of machines. This means that, if a class-machine mapping is used, only hierarchies of navigable associations can be translated and bi-navigable associations cannot be used.
3. Operations cannot call other operations within the same machine. This can be avoided by repeating the substitutions of the 'called' operation within the 'calling' operation in place of the call. The disadvantages of repeating blocks of substitutions can be avoided by using B definitions (a text substitution facility).

4. Simultaneous calls to several operations of another machine are not allowed. This means that if a one to one mapping between class methods and machine operations is used, class methods that simultaneously modify multiple instances of another class cannot be translated to valid B. This can be overcome by constructing a single operation of the associated class that alters the attribute values for multiple instances in a single substitution.

The majority of work in translating UML to B [LeDang and Souquières 2001; Lano et al. 2004] has started with the aim of translating each class to a B machine adding various strategies such as additional machines, to cater for the restrictions described above. This can lead to a complicated machine structure, which, although syntactically correct, is difficult to verify. In contrast, our approach concentrates on ease of proof. Initially we achieved this by restricting the UML class structures to those that can be mapped into B machines without contravening the restrictions mentioned above (i.e. only hierarchical tree structures of unidirectional associations could be used). In order to allow unconstrained association topologies, we have now developed a translation mapping, where a complete class diagram is translated into a single B component. This is the approach presented in this article. Structure is provided by the UML rather than by B. Semantics is provided in the form of attached constraints and action specifications, and proof and refinement are achieved via translation to B. Thus the limitations mentioned above are overcome.

1.5 The Object Oriented Approach

The UML is based on an object oriented modelling approach including the following key concepts: objects, encapsulation, class, generalization/specialization and messages. State machines can be used to model the behaviour of objects. We utilize these concepts as described below.

For our abstract systems modelling, objects are an abstraction of parts of a system that include state and events associated with those parts of the state. State is modelled by attributes, associations and state machines. Events are defined by a guard (a predicate on the state) that shows when they occur, and a substitution that shows how the state is changed by the event. For our modelling, object encapsulation is not important because the variables represent an abstract state, and we are only concerned with the effect operations have on the state, not how they achieve it. Sometimes events affect the state of other objects and we indicate this by specifying the change of value directly or by using a 'subroutine' of the other object. A form of encapsulation is provided by 'Packages,' which are used to modularize the model into, for example, sub models and refinements. We use classes to define sets of similar objects and 'specialization/generalization' relationships to show that the specialized class' objects are a subset of the generalized class' objects and have some additional or refined features.

Since we do not aim to 'modularize' a design using encapsulation, message passing between objects is not useful. The operations in our model represent events that occur spontaneously and may affect the state of several objects.

We use state machines to model the conceptual state of an object, which is additional to any state modelled by other variables of the object. Transitions in UML-B represent the behaviour of events of the objects and are atomic actions, thus avoiding many of the complications that can otherwise arise. This is effective for specification but may have limitations for implementation level modelling.

Modelling can be useful for many stages in the realization of a system component. We focus on abstract specification and early design stages.

1.6 Contribution

We present an approach based on UML, for modelling systems at an abstract level, adding detail through levels of refinement that can be verified against each other. The main contribution of our approach is that it results in a natural form of B, free of translation artefacts, that would otherwise complicate proof. This is achieved by translating multiple classes into a single B machine and avoiding operation calls. To support specification, we provide a systematic treatment of our action and constraint language, μB. Our method is supported by a tool that automates translation into B.

In this article, we extend previous work (e.g. chapter 6 in Mermet [2004]) by explaining the motivation for our work as well as its relationship to object orientation. We also introduce μB and its translation, subroutines, state dependent invariants, and state machine decision points, and provide an evaluation of the approach. In addition, all other parts are considerably improved with examples derived from an industrial case study.

1.7 Overview

In Section 2 we give an overview of the UML-B profile and μB action and constraint language. In Section 3 we present, with the help of examples from our case study work, our translation of UML-B class diagrams into B. In Section 4 we show how state machines can be used to add behavioral detail to these models. In Section 5 we show how UML-B models can be developed via refinement. In Section 6 we reflect on our experiences of using UML-B for industrial case studies. Section 7 summarizes and compares other people's work in similar translations. Section 8 concludes.

2. THE UML-B PROFILE

The UML-B is a profile of the UML, which defines a subset and specialization of UML that has a mapping to, and is therefore suitable for, translation into B language. The UML-B profile uses stereotypes to specialize the meaning of UML entities, thus enriching the standard UML notation and increasing its correspondence with B concepts. The UML-B profile defines tagged values (UML-B clauses) that may be used to attach details, such as invariants and guards, that are not part of the standard UML. Many of these clauses correspond directly with those of B, providing a 'fallback' mechanism for modelling directly in B when UML entities are not suitable. Other clauses are provided for adding

specific UML-B details to the modelling entities. UML-B provides a diagrammatic, formal modelling notation. Semi-formal UML features are given formality by their equivalence to B. Hence UML-B modelling is a completely formal process using translation to B for tool manipulation. UML-B hides B's infrastructure and packages mathematical constraints and action specifications into small sections, each being presented in the context of its owning UML entity.

2.1 μB—As an Action and Constraint Language

Initially, the UML concentrated on modelling the structural features of a design. Notations were provided for expressing functional behaviour at a requirements level, and state machines were available at lower levels, but the notations for expressing the behaviour of classes were incomplete. OCL can be used for expressing constraints on variable values within the model but a fully defined action notation is only now being introduced as part of UML 2.0. Many users were content to use incomplete models prior to the addition of code to implement behaviour. For our modelling, however, we require a complete behavioral model. We use a notation, μB (micro B) that borrows from B's abstract machine notation (AMN). μB has the following differences from AMN:- 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 may be omitted. For example, if i is omitted in the above, x refers to the value belonging to this. The reserved word this refers to the current contextual instance. (when μB is translated into B, this is translated into this<classname>, where <classname> is the name of the class). The symbol $ preceding a reference to a feature means the class-wide value of the feature (rather than the value for a single instance).

μB can be used to construct expressions which can then be used in predicates or substitutions based on the context of the containing class. Expressions can be used to evaluate an arithmetic, set, relation, or function value. Some examples of expressions are $S \cup T$, the union of sets S and T, and $R \lhd r$, the relation r restricted to only the set R as its domain (domain restriction). Further explanation and examples of μB expressions and a comparison with OCL are given in Section 3.

Predicates may use logic operators, such as conjunction, disjunction, implication, and quantification, set predicates such as membership and subset, and number predicates such as greater and less than. Expressions may also be used to construct substitutions that are used to specify actions. Some examples of μB are, $x := FALSE$, which sets the value of a Boolean attribute of the current instance; ANY v WHERE P THEN x:=v END selects any value that satisfies the predicate P and sets the value of the attribute, x, belonging to the current instance, to this value. Some of the commonly used elements of μB are summarized in Table I.

Table I. Summary of Commonly Used μB Elements

Substitutions	
S1 || S2	Parallel composition—S1 and S2 occur simultaneously
S1; S2	Sequential composition—S1 occurs followed by S2 (only allowed in refinements)
x := E	Assignment—after the substitution the value of x is the same as the expression E
SELECT P THEN S1 END	Guarded—if predicate, P, is true then S1 occurs
ANY x WHERE P THEN S1 END	Local variable, x, is given a value so that P is true and then S1 occurs
Predicates	
¬P	not
P1 ∧ P2	conjunction
P1 ∨ P2	disjunction
∀x · P	universal quantification
∃x · ¬ P	existential quantification
Basic Predicates	
E1 = E2	The expressions E1 and E2 have the same value
E1 ∈ E2	The expression E1 is a member of the set expression E2

2.2 UML-B Clauses

UML-B clauses provide a way of adding modelling information to the UML model that cannot be expressed diagrammatically. Each clause is a tagged value that can be attached to relevant entities. The UML-B profile defines the clauses that can be used via tagged values in this way. Any valid B clause (except OPERATIONS) has a corresponding meaning in UML-B although not all clauses are applicable with all modelling entities. For example, we use this method to specify invariants of a class. In addition to the usual B clauses, UML-B includes some clauses that extend UML to make alternative translation options available. The additional clauses are described later.

2.3 UML-B Model Architecture

Initially our strategy was to convert each class to a separate B component (i.e. machine, refinement or implementation) and to represent a B module (i.e. a machine and its refinements and implementation) as a UML package As previously mentioned the restrictions on class association topologies imposed by this class-component translation method became problematic for many industrial case studies. One possibility is to address these problems by collating features of classes into high level (controlling) machines where necessary but maintaining a class-machine translation as far as possible. However, for provability and traceability, we wished to keep a simple mapping from classes to B components so that the correspondence remained obvious, and to avoid creating translation artefacts (such as controllers) in the B. We chose packages as an alternative UML entity to represent a B component. The UML 'package' represents a collation mechanism for grouping class diagram modelling entities (such as classes and other packages) into a namespace. Packages, therefore, control visibility of other entities, without introducing additional semantics. In many ways packages are similar to the concept of B components, possibly more so than classes.

We therefore use packages in our UML models to represent model structural groupings. The top level package represents a complete system containing all its levels of detail. Packages contained within the top level package can represent either a B module (a module package) or a B component (a component package may be a machine or a refinement). To distinguish the intended meaning of a package we attach stereotypes to packages. Stereotypes are also used to control the interpretation of dependencies between packages.

3. CLASS DIAGRAMS

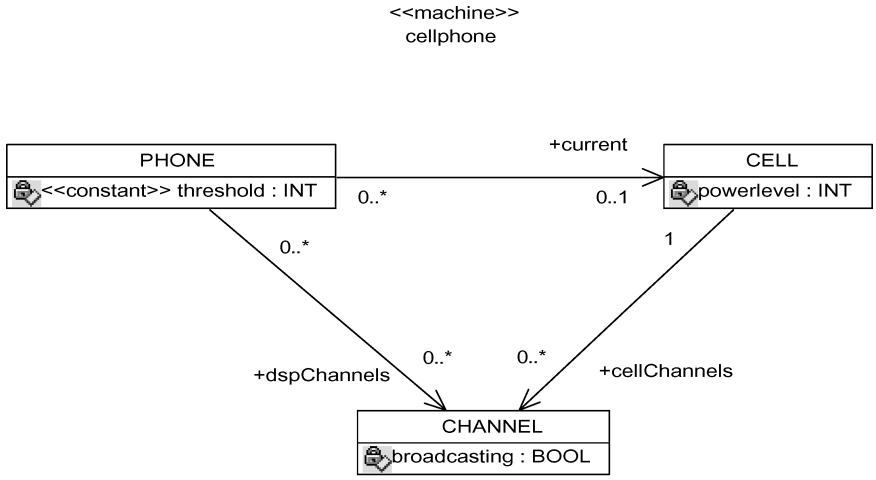
This section gives semantics to UML-B by describing the translation of basic class diagram features from UML-B to B including the representation of class instances, class data features and relationships between classes. We focus on conceptual modelling using class diagrams in an analysis phase. For the time being we do not support features, such as message passing, that would be needed in an implementation model. Classes are represented by B sets, constants, variables and operations and assembled into a single B component (i.e. machine or refinement, depending on the package stereotype). For example the model of a cellphone[5] and part of its translation into B is shown in Figure 3. The complete B translation of this example will be introduced and extended in subsequent sections.

The current set of instances of each of the three classes is represented by the variables PHONE, CELL and CHANNEL. These variables are defined in the type invariant as subsets of deferred sets (PHONE_SET, etc.) that represent the set of all possible instances for each class. The current instances sets are used as instance identities when referring to and manipulating the features (such as attributes) owned by a particular instance. Initially, no instances exist, hence the current instances sets are empty. (Note that we use B's macro facility, definitions, to structure the invariant. This is useful so that we can refer to parts of the invariant in predicates and for ensuring that the invariant is constructed in a valid order).

3.1 Attributes

In UML-B, a class represents a set of instances and class features, such as attributes, are implicitly replicated for each instance of the class. Since B is not object-oriented, this fundamental characteristic of object-oriented systems must be explicitly modelled. Hence, attributes are translated into variables whose type is a function from the instances set to the attribute type. The value of an attribute belonging to a particular instance can then be obtained by function application. For example, if x is an attribute of type T in class C, x is represented in the B model by a function mapping C to T, and the value for an instance, i, belonging to the class is given by x(i). Attribute types may be any μB set expression including predefined types (such as NAT, NAT1, BOOL and STRING), functions, sequences, powersets, instances of another class (referenced by the class name), and enumerated or deferred sets defined in a UML-B

[5]This example is a simplified version of a model developed with Ian Oliver of Nokia Research Centre, Helsinki.




```
MACHINE          cellphone
SETS             PHONE_SET ; CELL_SET ; CHANNEL_SET ; ...
DEFINITIONS
   type_invariant ==
                 (PHONE ∈ ℙ(PHONE_SET) ∧ CELL ∈ ℙ(CELL_SET) ∧
                     CHANNEL ∈ ℙ(CHANNEL_SET) ∧ ...);
   invariant ==  (type_invariant)
VARIABLES        PHONE, CELL, CHANNEL, ...
INVARIANT        invariant
INITIALISATION   PHONE:=∅ || CELL:=∅ || CHANNEL:=∅ || ...
```

Fig. 3. Cellphone—example of UML-B model and part of its translation in B.

SETS clauses. For example, the attributes of the cellphone example (Figure 3) are translated into B as follows.

```
MACHINE          cellphone
...
CONSTANTS        threshold
PROPERTIES       threshold ∈ PHONE_SET → INT
DEFINITIONS
   type_invariant ==   (... ∧
                 powerlevel ∈ CELL → INT ∧
                 broadcasting ∈ CHANNEL → BOOL ∧ ...);
   invariant ==  (type_invariant)
VARIABLES        ..., powerlevel, broadcasting
INVARIANT        invariant
INITIALISATION   ... || powerlevel := ∅ || broadcasting := ∅ ||
```

The attributes, powerlevel and broadcasting are represented by variables whose type is a function, and are initially empty as there are no instances in their domains. The attribute threshold is stereotyped as a constant (a stereotype defined in the UML-B profile). It is translated into a constant function from the possible instances set to its type. The values of constants are therefore

Table II. How Associations are Represented in B for Each Multiplicity Constraint

Association Representations in B for Different Multiplicities		
A *and* B *are the current instances sets of class* A *and* B *respectively,* f *is a function representing the association (i.e. the role name of the association with respect to the source class,* A). disjoint (f) *is defined in* B *as:* $\forall i,j \cdot (i \in dom(f) \wedge j \in dom(f) \wedge i \neq j \Rightarrow f(i) \cap f(j) = \varnothing)$		
UML association multiplicity	Informal description of B representation	B invariant
$0..* \rightarrow 0..1$	partial function to B	$f \in A \nrightarrow B$
$0..* \rightarrow 1..1$	total function to B	$f \in A \rightarrow B$
$0..* \rightarrow 0..*$	total function to subsets of B	$f \in A \rightarrow \mathbb{P}i(B)$
$0..* \rightarrow 1..*$	total function to non-empty subsets of B	$f \in A \rightarrow \mathbb{P}_1 (B)$
$0..1 \rightarrow 0..1$	partial injection to B	$f \in A \rightarrowtail B$
$0..1 \rightarrow 1..1$	total injection to B	$f \in A \rightarrowtail B$
$0..1 \rightarrow 0..*$	total function to subsets of B that don't intersect	$f \in A \rightarrow \mathbb{P}(B) \wedge$ disjoint(f)
$0..1 \rightarrow 1..*$	total function to non-empty subsets of B that don't intersect	$f \in A \rightarrow \mathbb{P}_1 (B) \wedge$ disjoint(f)
$1..* \rightarrow 0..1$	partial surjection to B	$f \in A \nrightarrow B$
$1..* \rightarrow 1..1$	total surjection to B	$f \in A \rightarrow B$
$1..* \rightarrow 0..*$	total function to subsets of B that cover B	$f \in A \rightarrow \mathbb{P}(B) \wedge$ union(ran(f))= B
$1..* \rightarrow 1..*$	total function to non-empty subsets of B that cover B	$f \in A \rightarrow \mathbb{P}_1 (B) \wedge$ union(ran(f))= B
$1..1 \rightarrow 0..1$	partial bijection to B	$f \in A \rightarrowtail B$
$1..1 \rightarrow 1..1$	total bijection to B	$f \in A \rightarrowtail B$
$1..1 \rightarrow 0..*$	total function to subsets of B that cover B without intersecting	$f \in A \rightarrow \mathbb{P}(B) \wedge$ union(ran(f))= B $\wedge$ disjoint(f)
$1..1 \rightarrow 1..*$	total function to non-empty subsets of B that cover B without intersecting	$f \in A \rightarrow \mathbb{P}_1 (B) \wedge$ union(ran(f))= B $\wedge$ disjoint(f)

preordained for all future instances of the class but may be different for each instance.

3.2 Associations

Associations are translated to functions in a manner similar to attributes except that the range of the function is based on the instances of the class at the supplier end of the association. Only associations that are navigable in one direction are used in UML-B. In UML, multiplicity ranges constrain associations. The multiplicities are equivalent to the usual mathematical categorizations of functions: partial, total, injective, surjective, and their combinations. Note that the multiplicity at the target end of the association (class B) specifies the number of instances of B that instances of the source end (class A) can map to and vice versa. The multiplicity of an association determines its modelling as shown in Table II. We use functions to subsets of the target class instances (e.g. $A \rightarrow \mathbb{P}(B)$) to model multiplicities with multiple targets. When the target multiplicity is at least one, $\mathbb{P}_1$ is used to ensure the subsets are non-empty.

For example, the associations of the cellphone example (Figure 3) are translated to the following B:

```
MACHINE          cellphone
...
DEFINITIONS
  disjoint(ff)== ∀a1,a2 · ( a1∈ dom(ff) ∧ a2 ∈ dom(ff) ∧
                                  a1≠a2 ⇒ ff(a1) ∩ ff(a2)=∅ );
  type_invariant ==  (... ∧
                      current ∈ PHONE ⇸ CELL ∧
                      dspChannels ∈ PHONE → ℙ(CHANNEL) ∧
                      cellChannels ∈ CELL → ℙ(CHANNEL) ∧ ...);
  CELL_invariant == (disjoint(cellChannels) ∧
                        union(ran(cellChannels)) =CHANNEL);
  invariant == (type_invariant ∧ CELL_invariant)
VARIABLES          ..., current, dspChannels, cellChannels
INVARIANT          invariant
INITIALISATION     ... ||current := ∅ || dspChannels := ∅ ||
                       cellChannels := ∅
```

The association, current, links a phone (but not all phones) with a single cell and is therefore a partial function. The other associations both link to zero or more channels and hence a total function to subsets of CHANNEL is used. For cellChannels, all channels are linked from exactly one cell and hence additional invariants are needed to ensure the sets of channels in the range are disjoint and cover all channels.

As for attributes, the stereotype, ≪constant≫ may be attached to an association or, if attached to a class, all class data is treated as constant. The stereotype, ≪static≫, may be attached to an attribute or association, meaning that it belongs to the whole class rather than a specific instance of the class. In this case instance mapping is suppressed giving a simple variable instead.

3.3 Translation of μB

Constraints and actions expressed in μB throughout the model must be converted to reflect the translation of state modelling features from object-oriented constructs into the set-based constructs available in B. We refer to this translation as T, where T(e) is the translation of the μB expression e into B.

If a μB expression contains a reference to a class feature, x, belonging to a specified instance, i, this would be written i.x in μB. As described in the previous section, the relationship between instances and the values of their features are represented in the corresponding B model by functions. Hence i.x is translated into function application, x(i). For example the predicate, $\forall_j.(j \in \forall \wedge j \in a \wedge x < j.y)$ attached to a class, C, tests an attribute, x, of the current instance to see whether it is less than the attribute, y, of all the linked instances in an association, a, to the class, A. Since this constraint is specified in the context of a class, it is implicitly a constraint applied to all instances of the class and during translation will be elaborated to $\forall$thisC. (thisC. $\in$ C $\wedge \forall_j$. (j $\in$ A $\wedge$ j $\in$ a(thisC) $\wedge$ x(thisC) $<$ y(j))).

If a is an association with a multiplicity greater than one at the target (supplier) end, then i is associated with a set of values. However, since the association is translated to a function from client instances to subsets of supplier

instances, the translation to function application is still valid. The features of an instance of an associated class may also be referenced using the dot notation transitively through a sequence of association links. For example if an instance, i of the current class is linked with an instance of another class via an association, a and that class has a feature, y, then the value of y for the instance associated with i can be referenced as i.a.y in μB. The expansion is, in correspondence with the direction of the associations, from the left so that i.a.y is equivalent to (i.a).y. This is translated by applying the function application translation twice, first for the feature, y and then for the feature, a. Hence i.a.y is translated via y(i.a) to y(a(i)). When the multiplicity allows zero target instances, it is important to ensure i has a link in the association (i.e. $i \in dom(a)$) otherwise a(i) is undefined. Note however, that y(a(i)) is only valid if a returns a single instance—if the association a has a multiplicity less than or equal to one. In future work we intend to strengthen the treatment of associations so that associations can be navigated more reliably whatever their multiplicities. For example, by translating e.a using relational image when the type of e is not singleton (i.e. T(u.a) is translated to a[T(u)] if T(u) gives a set.

In μB, (following the usual object-oriented style), the instance identifier (i in the above example) may be omitted from a reference to the value of a class feature. The reference has two different meanings, depending on where it occurs. When the reference is within an operation of the class, it refers to the value belonging to the instance for which the operation has been called (this). In the B model, a parameter, this<class_name>, of type, <class_name>, is added as a parameter of the operation, and the reference is translated to a(this<class_name>). When the reference is not associated with an operation, for example in an invariant attached to the class, the reference is implicitly generic for all instances of the class. In this case the same translation, a(this<class_name>), of a reference, a, is used and the complete expression is enclosed within a universal quantification for all instances of the class. For example if the μB expression, e contains such a reference to a feature of the class, C, $\forall thisC \cdot (thisC \in C \Rightarrow T(e))$.

We use μB instead of OCL because it is easier to relate information from the B proof tools (e.g. proof obligations and corrections) back to the UML-B models. It is also easier to translate from a syntax that is closer to our target notation; B. OCL has an operational style that is intended to be more accessible to programmers whereas μB has a more concise mathematical style, which lends itself to manipulation. OCL has a functional style, that results in asymmetric expressions. For example the union of two sets of instances would be written A.union(B) in OCL whereas μB has the more usual infix operator style, $A \cup B$. Selection of appropriate instances can be cumbersome in OCL, requiring the use of built-in functions, resulting in the main operation being obscured at the end of a long expression. For example, the following OCL expression finds the sum of an integer attribute, x, of class, C, for all instances that belong to a specialization, S, and are linked by the association, a.

context C:

def: myExpression: integer = $a \rightarrow$ select(oclIsTypeOf(S)).x $\rightarrow$ sum()

The corresponding expression in μB begins with the main sum operator, followed by a predicate to select the required instances and an expression to obtain the attribute to be summed.

$$\Sigma i \cdot (i \in C \wedge i \in S \wedge i \in a | i.x)$$

In many cases there is a high degree of correspondence and μB could be defined as an alternative concrete syntax for OCL. However, μB provides more flexibility, allowing the modeller to exploit the underlying semantic definition of data constructs (e.g. associations as functions) provided by U2B's translation. This can result in expressions that are more easily proved.

3.4 Behavior

Behavior is embodied in the specification of class operations and invariants using μB. This is illustrated in the examples below. If sequential composition of statements is avoided, the specification of behaviour in μB is equivalent to specification using OCL pre and post conditions.

3.4.1 *Invariant.* Invariants are specified using μB in UML-B INVARIANT clauses, which may be attached to various modelling entities. 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 two or more instances of a class). For instance invariants, the explicit reference to this may be omitted. The translation will add 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. The presence of explicit quantification is detected during translation. For example, if $x \in NAT$ is an attribute of class, C, then the following invariant could be attached to the class:

$$x < 100 \wedge \forall i,j \cdot ((i \in B \wedge j \in B \wedge i \neq j) \Rightarrow (i.x \neq j.x))$$

The first part, $x<100$, is an instance invariant because it applies to the attribute value for each and every instance of the class whereas the second part is a class invariant because it expresses a property that holds between the instances of the class. The invariant would be translated to:

$$\forall thisB \cdot (thisB \in B \Rightarrow (x(thisB) < 100)) \wedge$$
$$\forall i,j \cdot ((i \in B \wedge j \in B \wedge i \neq j) \Rightarrow (x(i) \neq x(j)))$$

The translation has added a universal quantification, $\forall thisB$, over all instances of B in the first part of the invariant. It is not used in the second part where the invariant already explicitly references instances of class B.

3.4.2 *Operation Specification.* Whereas in UML an operation may return a value of some given type, in B (and hence, UML-B), the types of multiple return values are inferred from the body of the operation. Hence UML-B deviates from UML and a list of the identifiers that represent operation return values is entered instead of a return type. In a similar way to attributes and

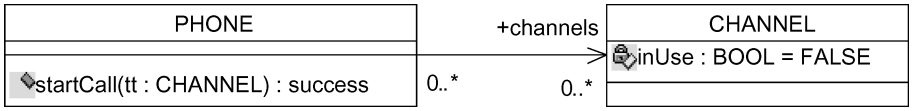


Fig. 4. Example of Operation Specification.

associations, class operations (unless ≪static≫ or ≪create≫) are implicitly performed using the data that belongs to a particular instance of that class. Hence operations need to know which instance of the class they are to work on. Since B is not object–oriented, operations must be explicitly associated with a particular instance of the class by adding a parameter, this<class_name>, of type, <class_name>, to each operation. This is used as the instance parameter in each reference to an attribute or association of the class. The instance parameter is inserted prior to any explicit parameters belonging to the operation. Details of operation behaviour are specified textually in μB guards and actions attached to the operation. Hence, an operation, o, with parameters, p1,p2,...,pn of types T1,T2,...,Tn and return variables r1,r2,..,rn will result in the following format B operation.

```
r1,r2,..,rn ← o(thisCLASS,p1, p2,...,pn) =
    PRE thisCLASS ∈ CLASS ∧ p1 ∈ T1 ∧ p2 ∈ T2 ∧...∧ pn ∈ Tn THEN
        SELECT <<guard>> THEN <<actions>> END
    END
```

The guard is a μB predicate involving any of the variables in the package. The action is a μB substitution that updates the values of variables (attributes, associations etc.) of the class via substitutions as described in Table I. If no guard or action is specified, the respective defaults are true (always enabled) and skip (do nothing).

In Figure 4, PHONE has an operation startCall that attempts to start a call on the channel †† and returns a Boolean representing its success. The call is successful if the channel is not already in use.

The operation, startCall, has a guard to ensure that the parameter, †† is a channel associated with the phone.

```
tt ∈ channels
```

The operation's action tests the inUse attribute of that channel:

```
IF tt.inUse = FALSE THEN tt.inUse := TRUE || success := TRUE
ELSE success := FALSE END
```

The guard and action are transformed and combined by U2B resulting in:

```
success ← startCall (thisPHONE,tt) =
PRE thisPHONE ∈ PHONE ∧ tt ∈ CHANNEL THEN
    SELECT tt ∈ channels(thisPHONE) THEN
        IF inUse(tt) = FALSE THEN inUse(tt) := TRUE || success := TRUE
        ELSE success := FALSE END
    END
END
```

3.4.3 *Initialization.* Initial values of variable class features may be specified either as specific values or as predicates to constrain a nondeterministic initialzation. The initial value field of attributes may be used to specify their initialzation. For other entities, such as associations, the UML-B clause, INITIALISATION or INITIALISATION_PREDICATE, may be attached to the association or owning class. For example, an integer attribute, x, could be initialized to 0 by attaching the clause, INITIALISATION x:=0, or initialized to any value less than 10 by INITIALISATION_PREDICATE x<10. For the latter case a convenient form of non deterministic substitution is provided in μB:

vars $\in$ (predicate) where vars is a comma separated list of variables.

This is equivalent to a substitution that sets all of the variables in vars so that the predicate is true. If no initial values are specified, the nondeterministic initialization, vars $\in$ (invariant) is provided by default. If any constraints on the initial values are provided in UML-B INITIALISATION_PREDICATE clauses, these are conjoined with the predicate:

vars $\in$(invariant $\wedge$ constraints)

3.4.4 *Appearance and Disappearance of Class Instances.* Create and destroy operations are useful in conceptual modelling when we wish to model events representing the spontaneous appearance and disappearance of objects in the system. The ≪create≫ stereotype can be attached to an operation of a variable instance class to indicate that it models an event where a new instance of the class appears. The operation will select an unused instance, initialise it as specified in the operation body (or on an attached state machine if it has a transition from the initial state with a matching event name). Create operations may be parameterised so that any of the class' variables (e.g. attributes, associations, state machines) are initialized, overriding any default initialzation values defined elsewhere in the class (such as attribute initial values or initial transitions on state machines). Similarly, the ≪destroy≫ stereotype can be used to indicate that the operation models an event where an instance disappears. The instance will be removed from the current instances set of the class and all mappings from that instance will be removed from the functions representing the variables of that class. Note that we are not concerned with garbage collection, although one use of ≪destroy≫ could be used to model the behaviour of a programme in this way.

3.4.5 *Subroutines.* The use of the package-component translation (i.e. all classes from a class diagram in a single B machine or refinement) was found to be more usable than the original class-component translation, but meant that there could be no method calling in the model. While this was found to be acceptable at an abstract level, as more detailed behaviour is added it is increasingly cumbersome to have to repeat common behaviour wherever it is needed. Also the design principles of encapsulation become more significant as the design progresses. To combat this limitation, the stereotype ≪subroutine≫ for class methods was introduced. Methods with this stereotype are translated into parameterized B definitions. Definitions are a literal, text substitution

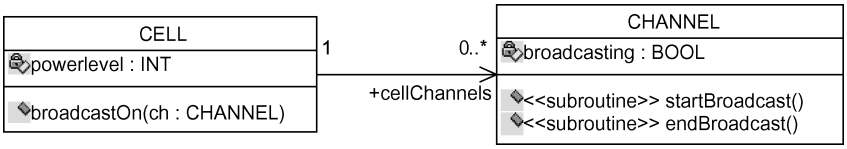


Fig. 5. Example of use of subroutines.

(macro) facility provided in B. Before a B component is type checked, each definition call is literally replaced by the definition body after substitution of the actual parameters. (Recursive definitions are not allowed.) Definitions can be instantiated wherever needed. The use of definitions in this way (which could become quite extensive if complex calling structures are modelled) has been found to be very effective. For example, in the cellphone model, a cell may need to initiate broadcasting on a particular one of its cellChannels. This could be achieved (Figure 5) by calling a subroutine, startBroadcast, which sets the attribute broadcasting to TRUE.

The following B definition would be produced to represent the subroutine. Also shown is an example of an operation using the subroutine.

```
DEFINITIONS
    startBroadcast (thisCHANNEL) ==
    BEGIN broadcasting(thisCHANNEL):=TRUE END;
    ...
OPERATIONS
    broadcastOn (thisCELL,ch) =
        PRE thisCELL ∈ CELL ∧ ch ∈ CHANNEL THEN
            SELECT  ch ∈ cellChannels(thisCELL)  THEN  startBroadcast(ch) END
        END ;
```

3.5 Specialization of Classes

Specialization represents subtyping of a class. (In programming languages, specialization is implemented via inheritance.) The instances of the subclass, being also instances of the superclass, retain all the variables (attributes, associations etc.) of the superclass but may add new variables that are only available to that subclass. Operation behaviour is retained by default, but may be overridden (i.e. re-defined) in a subclass. For example, the cellphone model is further developed in Figure 6 using specialization. A channel is one of three sub classes: an ACCESS channel, a TRAFFIC channel, or a CBCH channel. TRAFFIC defines a new attribute, callkind, which is only relevant to TRAFFIC channels. If the superclass is abstract (i.e. doesn't have instances other than those of its subclasses) then the subclass instances sets cover the set of super class instances. If B and C are disjoint subclasses of the abstract class A, then their instances would be modelled as $B \subseteq A \wedge C \subseteq A \wedge B \cap C = \emptyset \wedge B \cup C = A$. So far in our case studies we have not found a need for overlaid (i.e. non-disjoint) subclasses and the translation therefore assumes the subclasses are disjoint.

When the subclasses are translated into B, the current instances variable is defined as a subset of the superclass' current instances. Invariants are automatically added to ensure that the intersection between each pair of current

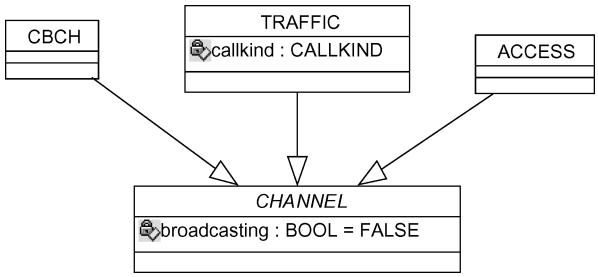


Fig. 6. Example of Specialization.

subclass instances is empty. The specialization defined in Figure 6 is modelled in B as follows:

```
REFINEMENT      cellphone1
REFINES         cellphone
SETS            CALLKIND={voice,data,other}
DEFINITIONS     ...
    type_invariant == (... ∧
                    CHANNEL ∈ ℙ(CHANNEL_SET) ∧
                    CBCH ∈ ℙ(CHANNEL) ∧
                    TRAFFIC ∈ ℙ(CHANNEL) ∧
                    ACCESS ∈ ℙ(CHANNEL) ∧ ... ∧
                    callkind ∈ TRAFFIC → CALLKIND ) ; ...
    package_invariant == (
                    CBCH ∩ TRAFFIC = ∅ ∧
                    CBCH ∩ ACCESS = ∅ ∧
                    TRAFFIC ∩ ACCESS = ∅ ) ;
    invariant == (type_invariant ∧ ... ∧ package_invariant)
VARIABLES       ..., CHANNEL, CBCH, TRAFFIC, ACCESS, ... , callkind
INVARIANT       invariant
INITIALISATION  ... || CHANNEL:=∅ || CBCH:=∅ || TRAFFIC:=∅ ||
                    ACCESS:=∅ ||... || callkind := ∅
```

4. BEHAVIOURAL SPECIFICATION BY STATE MACHINE

For some behaviour models a state machine representation is useful. A state machine can be attached to a class to describe its behaviour via a set of one or more state diagrams. The behaviour expressed in the state machine is combined with any μB operation specification. Hence operation behaviour can be defined either in μB or in a state machine or in a combination of both. The name of the state machine model represents a state variable. The collection of states in the state machine is an enumerated set that provides the type 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. State transitions define which operation changes the value of the state variable from the source state to the target state. This means that an operation is only available 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 event name must be given the same name as the operation. Substates are currently not supported but will be considered in future work.

If there is a transition from the initial state on the state machine, the target state of this transition is the initialzation value for the state variable. If there is a named transition from the initial state on the state machine, the state variable will be initialized in a ≪create≫ operation of that name attached to the class. Similarly, named final transitions will result in ≪destroy≫ operations, which remove the instance from the instance set.

Each transition has an implicit guard (the state being at its source) and an implicit action (to change the state to its target). Additional guard conditions (defined in μB) can be attached to a transition to further constrain when it can take place. Additional actions (also defined in μB) can be attached to transitions. The translator finds all transitions associated with an operation and compiles a guarded substitution of the following form:

```
SELECT statevar=sourcestate1 ∧ transition1_guards
THEN statevar:=targetstate1 || transition1_actions
WHEN statevar=sourcestate2 ∧ transition2_guards
THEN statevar:=targetstate2 || transition2_actions
....
END ||
```

The guarded substitution generated from the state machine is composed with the operation precondition and body μB specification (if any). If Po is the μB predicate in the operation guard, So the substitution from the operation actions and Gs the guarded substitution composed from the state machine, then the translator will produce the following operation:

```
SELECT Po THEN Gs || So END
```

Hence the μB guard is on the overall operation and, if false, the operation will not be enabled. In guarded simultaneous substitutions, S1 || S2, substitution cannot occur unless each simultaneous branch is enabled. This means that the textual operation semantics, although not associated with any particular state transition, is only enabled when at least one of the state transitions is enabled. Actions should be specified on state transitions when the action is specific to that state transition. Where the action is the same for all of that operation's state transitions, it may be specified in the operation body μB specification.

4.1 State Dependant Invariants

For many of our case studies we found that we needed to specify invariants concerning the value of attributes and associations while an instance of a class was in a particular state of a state machine. In many cases the state machine model is an abstract view of behaviour that is gradually replaced by a collection of other variables. During these refinements the correspondence of states to the values of the other variables must be indicated by such invariants. The INVARIANT clause may be used on a state to specify a predicate that should hold while an instances state variable is equal to that state. The hypothesis (state variable equals state) is automatically added to form the sequent. (Quantification over all instances will also be added as before.) Hence, for a class C, with state machine state, if the clause, INVARIANT p, is attached to a state S, then the

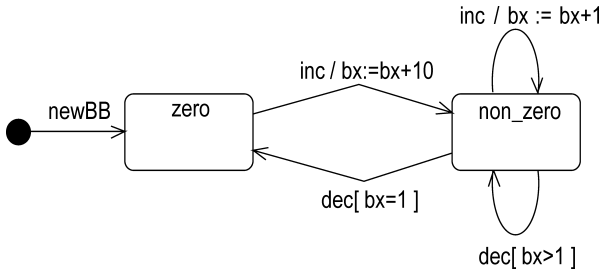


Fig. 7. Example of UML-B state machine.

following invariant would be generated in the B model:

$\forall thisC \cdot (thisC \in C \Rightarrow (state(thisC) = S \Rightarrow (T(p))))$

where T(p) is the translation of p from μB into standard B.

An example of the use of invariants on states is shown in the example below.

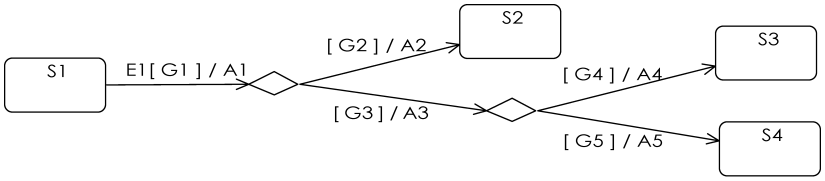
4.2 Example of State Machine Behaviour Specification

The example in Figure 7 illustrates how a state machine can be used to guard operations and define their actions and how common actions can be defined in the operation semantics window.

The state machine has two states, zero and non_zero. The implicit state variable, b_state (the name of the state machine) is treated like an attribute of type B_STATE = {zero,non_zero}. Invariants bx=0 and bx≠0 are attached to the states zero and non-zero respectively (not shown). When an instance is created its b_state is initialized to zero due to its initial transition. This state diagram results in the following B.

```
MACHINE          BB_CLASS
SETS             BB_SET;   B_STATE={zero, non_zero}
DEFINITIONS
   invariant == (
      BB ∈ ℙ(BB_SET) ∧ b_state ∈ BB → B_STATE ∧  bx ∈ BB → NAT ∧
      ∀thisBB · (thisBB ∈ BB ⇒ ((b_state(thisBB)=zero ⇒ (bx(thisBB)=0)) )) ∧
      ∀thisBB · (thisBB ∈ BB ⇒ ((b_state(thisBB)=non_zero ⇒ (bx(thisBB)≠0)) ))
                )
VARIABLES        BB, b_state, bx
INVARIANT        invariant
INITIALISATION   BB := ∅ || b_state := ∅ || bx := ∅
OPERATIONS
   Return ← newBB =
      ANY thisBB WHERE thisBB ∈ BB_SET-BB THEN
         BB := BB ∪ {thisBB} ||
         Return := thisBB ||
         b_state(thisBB) := zero ||
         bx(thisBB) := 0
      END
   END   ; ...
```

Operation inc can occur in either state. Its action is different depending on the starting state, hence actions have been defined on transitions and




```
E1 =  SELECT state= S1 ∧ G1 THEN A1 | |
      SELECT G2 THEN A2 | | state := S2
      WHEN G3 THEN A3 | |
          SELECT G4 THEN A4 | | state := S3
          WHEN G5 THEN A5 | | state := S4
          END
      END
END
```

Fig. 8. Illustration of state machine decision points and translation into B.

are combined with the state change action. This results in the following B operation:

```
inc (thisBB) =
PRE     thisBB∈BB THEN
    SELECT b_state(thisBB)=zero
    THEN   b_state(thisBB):=non_zero | | bx(thisBB):=bx(thisBB)+10
    WHEN   b_state(thisBB)=non_zero
    THEN   bx(thisBB) := bx(thisBB)+1
    END
END
```

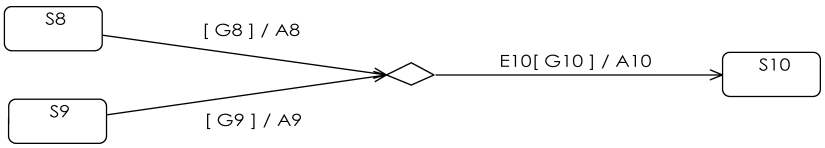
Operation dec has two guarded alternatives when in state non_zero but does not occur while in state zero. Since the action, bx := bx-1 is the same for both transitions it has been defined in the operation's μB actions specification rather than on a state transition. This results in:

```
dec (thisBB) =
PRE     thisBB∈BB THEN
    SELECT b_state(thisBB)=non_zero ∧ bx(thisBB)=1
    THEN   b_state(thisBB):=zero
    WHEN   b_state(thisBB)=non_zero ∧ bx(thisBB)>1
    THEN   skip
    END | |
    bx(thisBB):=bx(thisBB)-1
END
```

4.3 State Machine Decision Points

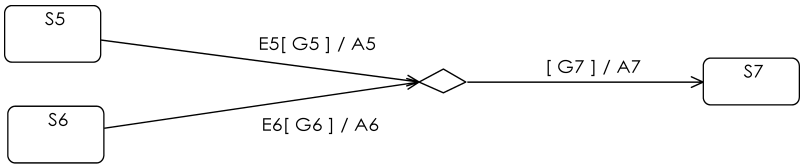
While using state machines we found that transition guards and actions can become overly complicated and are often partially repeated in several alternative transitions. To mitigate this we introduced the use of decision pseudo-states, which we use to structure sets of partially related transitions as shown below. Each decision point generates a SELECT substitution whose branches correspond to the outgoing transitions (Figure 8).

Decision pseudo states can also be used to merge several transitions so that the final transition represents an event with several alternative source states (Figure 11).




```
E10 = SELECT G10 THEN A10 || state:=S10 ||
      SELECT state=S8 ∧ G8 THEN A8
      WHEN  state=S9 ∧ G9 THEN  A9 END
END
```

Fig. 9. Illustration of merge point where event has several source states.




```
E5 = SELECT state=S5 ∧ G5 THEN A5 ||
     SELECT G7 THEN A7 || state :=S7 END
END
;
E6 = SELECT state=S6 ∧ G6 THEN A6 ||
     SELECT G7 THEN A7 || state :=S7 END
END;
```

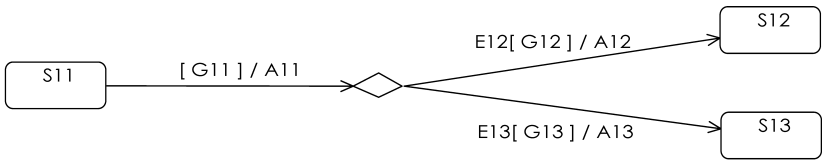
Fig. 10. Illustration of merge point used to share a final transition section.

Alternatively, the split or merge decision points can be used to share a transition section between several events (Figures 10 and 11).

Any combination of these uses of decision points can be constructed as long as every transition path that can be constructed from a source state to a target state includes one, and only one, transition segment that is annotated as an event.

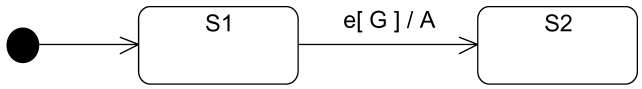
4.4 An Alternative Semantics for UML-B State Machines

While working on the PUSSEE project, an example was discovered where the expression of guards on state machine transitions was complicated and difficult to verify with the B tools. This led us to provide an optional alternative translation for state machines. Each state gives rise to a set containing the instances currently in that state. A transition is enabled (subject to other guards) if the current instance is a member of the starting state. The transition removes the instance from the starting state and adds it to the target state. The initial state contains all the instances of the class and the other states are empty. The state machine in Figure 12 produces two variables, S1 and S2, that are both of type $\mathbb{P}(C)$ (where C is the name of the class to which the state machine belongs). The invariant, $S1 \cap S2 = \{\} \wedge S1 \cup S2 = C$, ensures that the two sets are always disjoint. Initially, S1 contains all the instances, S1:=C || S2:={}. The transition event, e, is shown in Figure 12.




```
E13 = SELECT G13 THEN A13 || state:=S13 ||
      SELECT state=S11 ∧ G11 THEN A11 END
END
;
E12 = SELECT G12 THEN A12 || state:=S12 ||
      SELECT state=S11 ∧ G11 THEN A11 END
END
```

Fig. 11. Illustration of choice point used to share an initial transition section.




```
e (thisC) =
PRE thisC ∈ C THEN
    SELECT thisC ∈ S1 ∧ G   THEN
            S1:= S1- {thisC} || S2 := S2 ∪ {thisC} || A
    END
END
```

Fig. 12. Illustration of alternative state machine translation.

Although the change of state is slightly longer (involving changing two variables instead of one), with this semantics it is easier to express guards that depend on the state of other instances. For example, S1={this} would enable the transition only when there are no other instances in S1. On the other hand, it can be more cumbersome to determine the current state of a given instance since this involves testing the membership of the instance against each state set. This alternative semantics should be used when it is necessary to refer extensively to the set of instances in particular states in an invariant or guard.

5. REFINEMENT

The B method is based on a hierarchical stepwise refinement and decomposition of a problem. After initial informal specification of requirements, an abstraction is made to capture the most essential properties of a system. For example, these could be the main safety properties in a safety critical system. This abstract specification is made more concrete and detailed in steps of two types. The specification can be refined by changing the data structures used to represent state information or the operations that act upon these data structures. Alternatively, the specification can be decomposed into subsystems via an implementation step that binds the previous refinement to one or more abstract machines representing the interfaces of the subsystems. In a typical B project many levels of refinement and decomposition are used to fully specify the requirements. Once a stage is reached when all the requirements have been

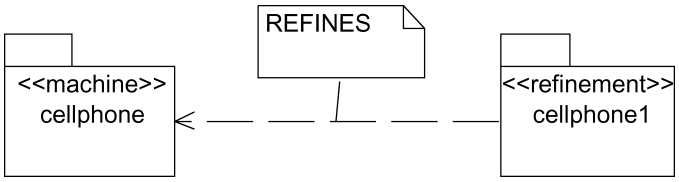


Fig. 13. Using a package to indicate refinement.

formally expressed, further refinement and decomposition steps add implementation decisions until a level of detail is reached where code can be generated. At each refinement or decomposition step, proof obligations are generated and must be discharged in order to prove that the outputs of the step are a valid refinement of the previous level. At each step when more detailed requirements are introduced or implementation steps are taken, it is proved that they respect all the previous levels. This method ensures that the developed system obeys the properties expressed in all the levels of specification from which it is derived. Such proof is not always easily achieved. While the tool automatically discharges most proof obligations, typically some 20% require human interaction [ClearSy 2000] and this interactive proof requires expertise and effort. The form and style of the formal B specification can greatly affect the ease of achieving these proof obligations. Hence ease of proof rather than any design paradigm becomes the primary criterion for developing specifications in B. This is why refinement and decomposition are the significant mechanisms in building a B specification. A mechanism for structuring a specification within a refinement level is provided (INCLUDES). This can be useful for segregating and encapsulating state data and its associated behaviour to aid understanding, but contributes less to ease of proof.

5.1 Refinement in UML-B

Since our aim is to reflect the B method in our UML-B notation, we cater for abstraction-refinement concepts in our UML-B models. We have purposely maintained a simple correspondence between UML entities and B components so that this is easily achieved. The stereotypes, ≪machine≫ and ≪refinement≫, used to control the translation, identify the UML entities (packages) that are involved in the refinement structure. The entity refined by a refinement is indicated by a UML-B REFINES clause. There are several differences in the translation of refinements from those of machines. For example, the heading generated in the B component is different—a REFINES clause is added—and variables with the same names as those in the abstraction are assumed to have the same type. For example, in Figure 13, the cellphone model of Figure 3 is refined by a more detailed model in a new package, cellphone1.

The refinement (Figure 14) uses specialization to introduce three subclasses of the CHANNEL class. The class is split into the subclasses CBCH, TRAFFIC and ACCESS, which will be represented by subsets of the CHANNEL instances set, as discussed in the section on specialization. This specialization results in a corresponding refinement to the associations, channels and cellChannels,

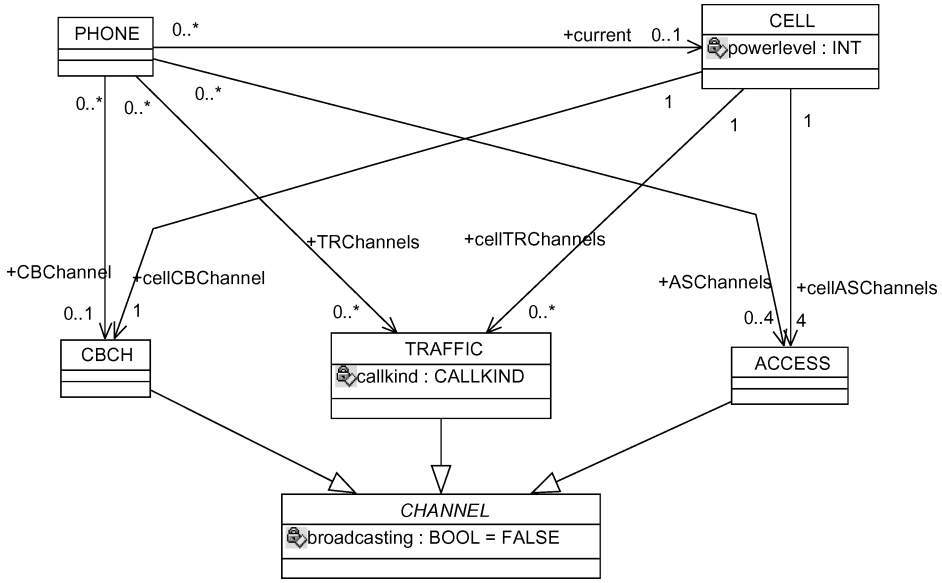


Fig. 14. Class diagram for refinement, cellphone1.

Table III. Refinement Relation for Cellphone1

Class	REFINEMENT_RELATION
PHONE	channels = {CBChannel} ∪ TRChannels ∪ ASChannels
CELL	cellChannels = {cellCBChannel} ∪ cellTRChannels ∪ cellASChannels

that previously linked PHONE and CELL to the CHANNEL class in the abstract package. These associations are each refined by three new associations that link to the three new subclasses.

In a B refinement, part of the invariant describes the relation between the variables of the refinement and those of the abstraction that they refine. This relationship is a special kind of invariant in addition to the internal constraints of the component itself. In UML-B, it is useful to distinguish the refinement relation from the rest of the invariant by providing a separate UML-B clause called REFINEMENT_RELATION. There are translation motivations for separating the refinement relation. The invariant may be used in an initialzation predicate whereas the refinement relation should not be used in this way. (This is because the variables of the abstraction are not visible anywhere other than in the B INVARIANT clause.) For the cellphone1 example, the refinement relation, shown in Table III, specifies that, for each phone, the set of channels given by the abstract association channels, is formed from the sets of subclass instances given by the three new associations, CBChannel, TRChannels and ASChannels. Since CBChannel has multiplicity 0..1, it is enclosed in set brackets to make the single instance into a set.

As shown in Figure 13, the refinement relation is translated into B by adding universal quantification over instances of the classes, PHONE and CELL.

```
REFINEMENT      cellphone1
REFINES         cellphone
...
DEFINITIONS  ...;
    type_invariant == (...) ;
    PHONE_invariant == (...) ;
    CELL_invariant == (...) ;
    package_invariant == (...) ;
    invariant == (type_invariant ∧ PHONE_invariant ∧
                  CELL_invariant ∧ package_invariant);
    refinement_relation == (∀thisPHONE • (thisPHONE∈PHONE ⇒ (
        dspChannels(thisPHONE)= {dspCBCH(thisPHONE)} ∪
            dspTrafficChannels(thisPHONE) ∪ dspASChannels(thisPHONE) )) ∧
        ∀thisCELL • (thisCELL∈CELL ⇒ ( cellChannels(thisCELL)=
            {cellCBCH(thisCELL)} ∪ cellTrafficChannels(thisCELL) ∪
                cellASChannels(thisCELL) )) )
VARIABLES       ...
INVARIANT    invariant ∧ refinement_relation
```

Fig. 15. B model for refinement, cellphone1.

The cellphone example illustrates the contribution that UML-B brings to B: refinements are expressed on instances that are automatically lifted to classes (similar to promotion in Z). This, and the visualization of corresponding features in the refined models, make UML-B refinement easier to describe and understand than pure textual B. The contribution to UML is the notion of a refinement relation between abstract and concrete models and the ability, based on B refinement, to verify that relationship.

6. DISCUSSION AND EXPERIENCE

Several case studies using UML-B are reported in Mermet [2004] including an adaptive cruise controller for a vehicle, a hardware device for a serial communications link, the architecture for a GSM (Global System for Mobile communications) device and a hamming coder/decoder. A UML-B based hardware/software codesign process for designing embedded systems is described in Voros et al. [2004]. UML-B was used successfully to formally develop the safety requirements for a real-time control system [Snook et al. 2003]. This example concentrated on the refinement of state machine models using the transition decision points to refine transitions. The case study was successfully proven using the AtelierB prover throughout several levels of refinement as the system model was decomposed in subsystems. The case study also highlighted how hierarchical states could be used as a natural form of refinement of state data. In a refinement, a state of the abstract model can be given further behavioral detail by turning it into a super-state with a sub-statemachine. The new states represent additional data and the transitions between them represent new events (i.e. refine 'do nothing'). In future work we intend to develop these aspects of UML-B to support refinement.

During these case studies, industrial partners that were not experienced in the use of formal methods, found UML-B provided an accessible route into using

formal specifications. In particular, confidence that the requirements are valid, was improved. The reaction from formal methods experts was mixed. They thought the UML-B specification would be useful documentation for customers but some viewed B as the primary modelling notation.

Some experienced B users commented that UML-B hides information that would be available in the B text. Our view is that UML-B diagrams are clearer and better structured, but are just as precise as B, and their semantics are well defined via the translation to B. For example, the nature of an association (relationship) between two classes (sets) defined by its multiplicities is more readily envisaged. The juxtaposition of classes and associations is clearly displayed so that the intention of the model constructs is more easily deduced. However, we can only go so far with diagrams. Textual specification is needed for details of constraints and actions. When constraints and actions are expressed textually in μB they are made more visible by hiding B infrastructure, allowing their significance and context to be highlighted. UML structures a model into a hierarchical system of views. While this is often useful to aid clarity it is sometimes true that bits of information are difficult to locate. For example it would be useful to be able to see invariants on diagrams. Currently we are working on a UML-B drawing tool that allows better access to undisplayed parts of the specification.

Since the verification tools are currently B based (rather than UML-B based) it is important that the B view is available and is readable. Many people have commented on this fact and asked how the corrections can be traced back to the UML-B. In practice we have found that it is easy to locate the relevant components in the UML-B. This vindicates the criterion of our approach, which is to maintain a simple mapping from UML components to B components in order to ensure that the B-based verification tools are practically usable. Several organizations, industrial and academic, have indicated agreement with this philosophy. The overall lesson is that proof is not easy (even with semi-automatic provers) and if it is to be achieved, consideration must be given to provability in generating the models (whether writing B by hand or translating from UML-B). Proof is an important issue and, in the future, we hope that the verification tools will be better integrated with UML-B. Ideally, the tools would work directly on the UML-B model and provide error and proof information in terms of this model. An intermediate stage that would be more easily achievable is for the tools to work with B but provide feedback into the UML-B to illustrate the errors.

7. OTHER WORK ON TRANSLATING TO B

Several groups have proposed translations from object-oriented notations to B. As well as those discussed below, see earlier work by Nagui-Raiss [1994] and Shore [1996]. Suggestions for modelling static class data and relationships are similar to each other and were originally the basis for our approach. However, due to the difficulties discussed above, of representing an object-oriented model these approaches generally result in B that is overly complex and this may damage our goal of providing usable validation and verification. Our approach

differs because our aim is to provide a UML based formal modelling notation that facilitates verification by proof. Whereas most groups attempt to translate without interfering with the UML representation, we provide a profile that allows the modeller to control the translation.

Lano et al. [2004] describe translation of UML based specifications, including OCL constraints, into B. They arrange a structure of machines using the INCLUDES and USES constructs to cope with interclass interactions. In our experience such structures can make verification more difficult. Facon et al. [1996] provide a mapping of class diagram features into B machines. Their work has concentrated on information systems and database applications that are data-centric [Facon et al. 1999]. These types of systems involve a high degree of data modelling but only simple operations. Consequently, our behaviour modelling would be largely redundant. Their approach is to automatically generate basic operations according to class properties such as mutability and multiplicity. Class state machines are then used to define how external events invoke the basic operations of the class according to state and guard conditions. Collaboration diagrams define which class events occur in response to system transactions. Thus, the hierarchy of system behaviour is represented in layers made up of different UML modelling notations (collaboration, state, and class) rather than by reflecting hierarchy in the model as we do. This approach is more suited to data intensive systems whereas our approach supports more general modelling of systems.

Meyer and Souquières [1999] propose a method for transforming OMT diagrams (on which UML class diagrams are based). Classes are provided with basic operations and a class state machine adds functionality by defining events and state transitions under which these basic operations are used. The state machine layer is represented as operations within the class machine. To avoid calling operations within the same machine, basic operations are translated to definitions rather than operations. The resulting structure of B machines consists of a top-level system machine, a machine for each class (including subclasses and aggregate components), and a machine for each unfixed (or attributed) association. A disadvantage is that some class behaviour is elevated to the top-level machine in order to obtain write access over association links. In further work by Ledang and Souquières [2001] the calling sequence defined in a collaboration diagram is used to construct a structure of B machines with one machine for each layer except at the bottom layer, where there is one machine for each class. Implementations and imports are used to overcome operation calling restrictions.

Sekerinski [1998] describes how reactive systems can be designed graphically using statecharts [Harel 1987] and how these designs can be converted to B for analysis and refinement to code. The treatment differs from ours in that statecharts, although similar to UML state machines, are treated as an independent form of design notation rather than as a subnotation to class diagrams. Hierarchical statecharts (i.e. states may have substates) and concurrency (i.e. states may have groups of substates that may progress independently and concurrently) are included. These are areas that we are currently addressing.

8. CONCLUSIONS

We have found that UML-B can be used to model a variety of problem types at different levels of abstraction using its different modes and semantic options. We have found our strategy, which converts the contents of a complete package into a single B component, to be more useful than the previous class-component translation. For example, the cellphone model in Figure 3 could not have been translated using the class-component translation. This strategy has allowed us to create useful UML-B models that can be translated into B in a style that is amenable to the proof tools. We have validated our approach on a variety of nontrivial industrial problems in cooperation with industrial partners. We have yet to test its scalability on very large problems, but our expectation is that it will scale in the same way that B scales, through refinement and decomposition. A key to achieving this will be robust tool support with rich functionality. In future work we will continue to develop UML-B in close cooperation with industrial partners and with ongoing developments in the B language.

The precise semantics given to UML-B and the emphasis on using existing B proof techniques and tools should be of interest to the UML community, especially those concerned with précising the semantics of UML and those concerned with applying UML to critical system design.

ACKNOWLEDGMENTS

Contribution from industrial and academic users has been essential in the development of a usable UML-B modelling method and we would like to acknowledge the contributions of our partners in the MATISSE and PUSSEE projects. In particular, Ian Oliver of Nokia Research Centre, Helsinki; Stefan Hallerstede of Keesda, Grenoble; Ola Lundkvist of Volvo, Gothenburg and Marina Waldén of Åbo Akademi, Turku. We also acknowledge the comments of the anonymous TOSEM reviewers, which lead to improvements in the article.

REFERENCES

ABRIAL, J. R. 1996. *The B Book—Assigning Programs to Meanings.* Cambridge University Press, NY.

AMEY, P. 2004. Dear sir, yours faithfully: an everyday story of formality, In *Practical Elements of Safety: Proceedings of the twelfth Safety-Critical Systems Symposium*, Birmingham, UK, February, F. Redmill and T. Anderson, Eds. Springer-Verlag, London, 3–15.

B-CORE 1996. *B-Toolkit User's Manual, Release 3.2.* B-Core(UK) Ltd., Oxford, UK.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. 1999. *Model Checking.* MIT Press, Cambridge, MA.

CLEARSY 2003. *AtelierB User Manual V3.6.* ClearSy System Engineering, Aix-en-Provence, F.

CLEARSY 2000. *AtelierB Training Course Level 2*. ClearSy System Engineering, Aix-en-Provence, F.

CRAIGEN, D., GERHART, S., AND RALSTON, T. 1995. Formal methods reality check: Industrial usage. *IEEE Trans. Softw. Eng. 21*, 2, 90–98.

FACON, P., LALEAU, R., AND NGUYEN, H. P. 1996. Mapping object diagrams into B specifications. In *Methods Integration Workshop, Electronic Workshops in Computing (eWiC),* Leeds, UK, March. Springer-Verlag.

FACON, P., LALEAU, R., NGUYEN, H. P., AND MAMMAR, A. 1999. *Combining UML with the B Formal Method for the Specification of Database Applications*. Research report, CEDRIC Laboratory, Paris, F.

GLASS, R. 2004. The mystery of formal methods disuse. *Comm. ACM 47*, 8, 15–17.

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Prog. 8*, 3, 231–274.

LANO, K., CLARK, D., AND ANDROUTSOPOULOS, K. 2004. UML to B: Formal Verification of Object-Oriented Models. In *Integrated Formal Methods, 4th International Conference, IFM 2004,* Canterbury, UK, April 2004, E. A. Boiten, J. Derrick and G. Smith, Eds. Lecture Notes in Computer Science Vol. 2999, Springer-Verlag, Berlin Heidelberg, 187–206.

LEDANG, H. AND SOUQUIÈRES, J. 2001. Integrating UML and B specification techniques. In *Informatik 2001 Workshop on Integrating Diagrammatic and Formal Specification Techniques.* Vienna, Austria, September. GI Jahrestagung (1) 2001, 641–648.

LEUSCHEL, M. AND BUTLER, M. 2003. ProB: A Model-Checker for B, In *Proceedings of Formal Methods Europe, FME 2003,* Pisa, Italy, A. Keijiro, S. Gnesi and M. Dino, Eds. Lecture Notes in Computer Science Vol. 2805, Springer-Verlag, Berlin Heidelberg, 855–874.

MATISSE 2003. Methodologies and Technologies for Industrial Strength Systems Engineering (MATISSE) *IST Programme RTD Research Project IST-1999-11435.* 1 May 2000 to 28 February 2003 http://www.matisse.qinetiq.com/

MERMET, J. (ED.) 2004. *UML-B Specification for Proven Embedded Systems Design*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

MEYER, E. AND SOUQUIÈRES, J. 1999. A systematic approach to transform OMT diagrams to a B specification. In *World Congress on Formal Methods in the Development of Computing Systems, FM'99, Vol. I*. Toulouse, France, September 1999, J. Wing, J. Woodcock and J. Davies, Eds. Lecture Notes in Computer Science, Vol.1708, Springer-Verlag, Berlin Heidelberg, 875–895.

NAGUI-RAISS, N. 1994. A formal software specification tool using the entity-relationship model. In *13th International Conference on the Entity-Relationship Approach*. Manchester, U.K., December, P. Loucopoulos, Ed. Lecture Notes in Computer Science, Vol. 881, Springer-Verlag, 315–332.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 1998. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, MA.

SEKERINSKI, E. 1998. Graphical design of reactive systems. In *B'98: Recent Advances in the Development and Use of the B Method—2nd International B Conference.* Montpellier, France, April 1998, D. Bert, Ed. Lecture Notes in Computer Science, Vol.1393, Springer-Verlag, 182–197.

SHORE, R. 1996. Object-oriented modelling in B. In *Proceedings of the 1st Conference on the B method.* Nantes, France, November, H. Habrias, Ed. 133–154.

SNOOK, C., OLIVER, I., AND BUTLER, M. 2004. The UML-B profile for formal systems modelling in UML, In *UML-B Specification for Proven Embedded Systems Design*, J. Mermet, Ed. Kluwer Academic Publishers, Dordrecht, The Netherlands.

SNOOK, C. AND BUTLER, M. 2004. U2B—A tool for translating UML-B models into B, In *UML-B Specification for Proven Embedded Systems Design*, J. Mermet, Ed. Kluwer Academic Publishers, Dordrecht, The Netherlands.

SNOOK, C., TSIOPOULOS, L., AND WALDÉN, M. 2003. A case study in requirement analysis of control systems using UML and B, In *Proceedings of RCS'2003 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience.* Turku, Finland, June.

SPIVEY, J. M. 1988. *Understanding Z a specification language and its formal semantics*. Cambridge University Press, NY.

TOVAL, A., REQUENA, A., AND ALEMÁN, J. L. 2003. Emerging OCL Tools. *Software and System Modeling (SoSyM), 2*, 40, 248–261.

VAZIRI, M. AND 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. Available at http://sdg.lcs.mit.edu/cdnj/publications.

VOROS, N., SNOOK, C., HALLERSTEDE, S., AND MASSELOS, K. 2004. Embedded System Design Using Formal Model Refinement: An Approach Based on the Combined Use of UML and the B Language, *Design Automation for Embedded Systems 9*, 2, 67–99.

WARMER, J. AND KLEPPE, A. 2003. *The Object Constraint Language Second Edition: Getting your models ready for MDA.* Addison-Wesley.

Received December 2004; revised July 2005; accepted November 2005