Softw Syst Model (2015) 14:1557–1580
DOI 10.1007/s10270-013-0391-z

REGULAR PAPER

A method of refinement in UML-B

Mar Yah Said · **Michael Butler** · **Colin Snook**

Received: 20 May 2012 / Revised: 25 August 2013 / Accepted: 10 November 2013 / Published online: 14 December 2013

Abstract UML-B is a ‘UML-like’ graphical front-end for Event-B that provides support for object-oriented and state machine modelling concepts, which are not available in Event-B. In particular, UML-B includes class diagram and state machine diagram editors with automatic generation of corresponding Event-B. In Event-B, refinement is used to relate system models at different abstraction levels. The same refinement concepts are also applicable in UML-B but require special consideration due to the higher-level modelling concepts. In previous work, we described a case study to introduce support for refinement in UML-B. We now provide a more complete presentation of the technique of refinement in UML-B including a formalisation of the refinement rules and a definition of the extensions to the abstract syntax of UML-B notation. The provision of gluing invariants to discharge the proof obligations associated with a refinement is a significant step in providing verifiable models. We discuss and compare two approaches for constructing gluing invariants in the context of UML-B refinement.

Keywords Visual modelling languages · Formal specification · UML-B · Event-B · Class diagram · State machine

Communicated by Dr. Kevin Lano.

M. Y. Said (✉)
FSKTM, Universiti Putra Malaysia, Serdang, Malaysia
e-mail: maryah@upm.edu.my

M. Butler · C. Snook
ECS, University of Southampton, Southampton, UK
e-mail: mjb@ecs.soton.ac.uk

C. Snook
e-mail: cfs@ecs.soton.ac.uk

1 Introduction

UML-B [32] is a graphical front-end for Event-B [22] that has some similarity with UML [24,27]. Event-B is a state-based formalism with support for refinement and proof. UML-B supports class and state machine diagrams, concepts that are not supported in Event-B. UML-B was originally based on B [1] (now often referred to as ‘Classical B’), which is a method for verified software development based on the Abstract Machine Notation (AMN) and set theory. The UML-B tool, supporting the current, Event-B-based, UML-B notation, is a plug-in to the Rodin platform [3,7]. Event-B models are generated from UML-B models by the UML-B tool. The Rodin tools are used to report back any static verification errors and then generate and prove proof obligations associated with the generated Event-B models. The purpose of UML-B is twofold. Firstly, it provides two essential modelling concepts that are absent in Event-B, object orientation and event sequencing. Secondly, it provides a more approachable and efficient modelling notation, especially for those familiar with UML.

Event-B was developed as an alternative to Classical B in order to support modelling at a system’s level. Event-B distinguishes between contexts and machines so that machines can be reused with different configurations. A context contains static configuration including given sets, constants and properties (including type) of the constants. A machine, which may *see* several contexts, contains state variables, invariant properties of the variables and events that update the variables. Each event is a guarded action which may fire when its guard is true and then executes actions, which change the state of the variables. The guards of an event are predicates over the variables of the machine and any local parameters of the event. The actions are a set of parallel substitutions which change the state of the variables.

Refinement [1,2] is the process of building a model gradually by making it more and more precise while verifying that each refined model satisfies the abstract behaviour. The advantage of starting with an abstract model is that important properties can be defined in a simple model, which is therefore less likely to contain mistakes. Refinements then introduce detail in steps which are guaranteed to preserve the important properties. Refinement in Event-B involves refining the model state by adding or substituting variables and refining events into corresponding concrete versions and by adding new events. The abstract state variables, x, and the concrete state variables, y, are linked together by a predicate called a gluing invariant $J(x, y)$. The gluing invariant provides the link between the abstract and concrete representations of state that is needed to verify that each abstract event is a correct simulation of its concrete version. Providing sufficient but provable gluing invariants can be a significant task. The refinement concepts of Event-B must be supported in UML-B for it to be a successful front-end to Event-B.

An alternative step to refinement is decomposition of a model into parts, which is important for scalability. Mechanisms for decomposing Event-B models that retain composition of verification have been developed, and work to support these in UML-B is currently underway. Decomposition is not covered in this paper.

Our previous work in [28] detailed a case study that was used to investigate support for class and state machine refinement in UML-B. We now provide a more complete presentation of the technique of refinement in UML-B. The contributions of this paper are as follows:

- a description of the intuition behind UML-B refinement,
- a formalisation of the UML-B refinement rules using the Event-B notation,
- a definition of the extensions to the abstract syntax (meta-model) of the UML-B notation that were needed to support refinement,
- a discussion on constructing gluing invariants in the context of UML-B refinement.

The organisation of the paper is as follows. In Sect. 2, we give some background on UML-B and the generated Event-B excluding refinement. Section 3 gives an intuitive overview of UML-B refinement in terms of data refinement and event refinement. Section 4 gives a formalisation of the syntactic rules for class diagram and state machine refinement. Section 5 describes the extensions to the abstract syntax (i.e. meta-model) of the UML-B notation. Section 6 provides an overview of the ATM case study development. Section 7 discusses two alternative approaches to constructing gluing invariants. Section 9 concludes the paper. Section 8 presents some related work.

2 Background of UML-B and generated Event-B

There are four kinds of diagrams in UML-B. They are package, context, class and state machine diagrams. A package diagram shows the structure and relationships between components (machines and contexts) in a project. A context diagram is similar to a class diagram but contains constant data and structured types. A context diagram described a context. A machine is specified by a class diagram and state machine diagram(s) representing data structures that may be changed by events or transitions. Events may be attached to classes in a class diagram. Events can also be represented by the transitions in a state machine diagram. The semantics of a UML-B model is given by the Event-B generated by the UML-B tool according to a set of translation rules. The following subsections describe more about the package, context, class and state machine diagrams.

2.1 Package diagram

A package diagram defines the relationships between UML-B machines and contexts in a project. Figure 1 shows an example of a package diagram. The package diagram consists of machines *M1* and *M2* and refinement relationship between them. It also contains contexts *CX1* and *CX2* and the extension relationship between them and which contexts are seen by the machines.

2.2 Context diagram

A context diagram defines the static part of a model. A context diagram may have classtypes. Each classtype may have attributes and associations. An attribute defines a constant that has a data value for each instance of a classtype. An association is a special case of an attribute that defines a relationship between two classtypes. Figure 2 shows an example of a context diagram. The classtype *CUSTOMER* has an attribute *ident* and an association *accounts*, with the classtype *BANK*. The 1..1 target multiplicity of the association *accounts* indi-

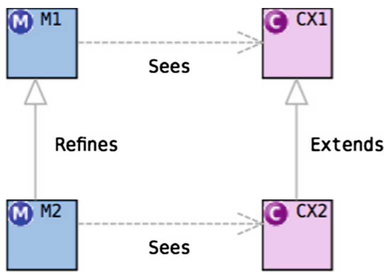
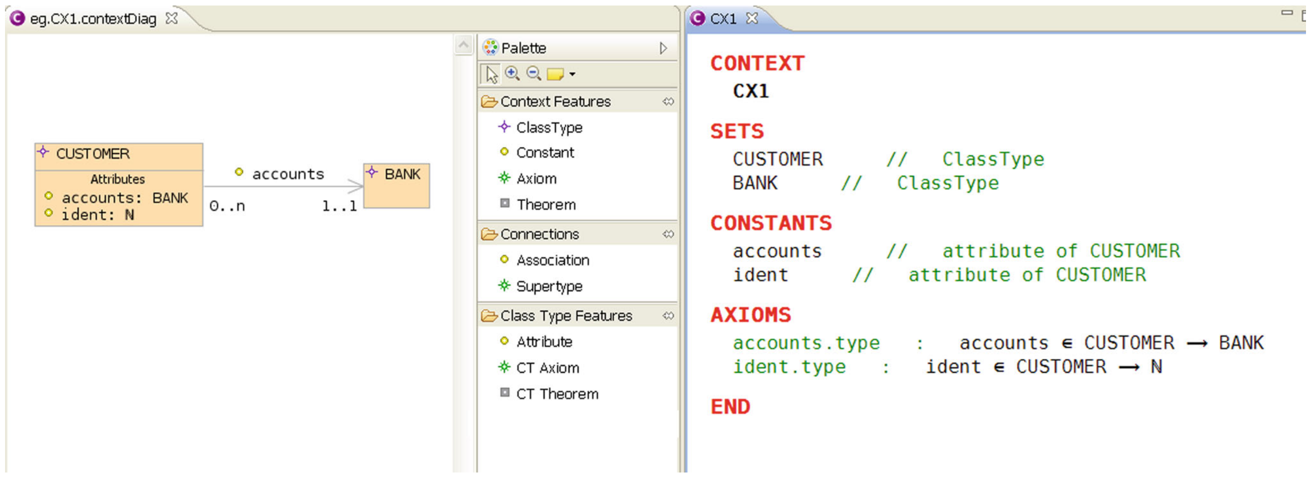


Fig. 1 Example of package diagram



```
CONTEXT
  CX1
SETS
  CUSTOMER     //   ClassType
  BANK     //   ClassType
CONSTANTS
  accounts     //   attribute of CUSTOMER
  ident     //   attribute of CUSTOMER
AXIOMS
  accounts.type   :   accounts ∈ CUSTOMER ⟶ BANK
  ident.type   :   ident ∈ CUSTOMER ⟶ N
END
```

Fig. 2 Examples of context diagram and Event-B translation

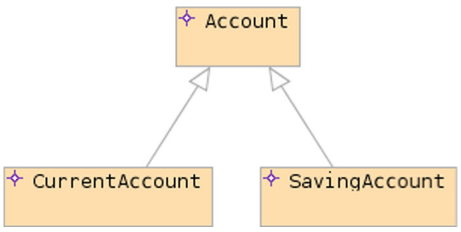


Fig. 3 Subtyping a classtype in UML-B

cates that it is a total function. Axioms and theorems[1] may be attached to a classtype in which case the predicate must be true for all instances of the classtype. Classtypes are used to define types and also to define constant attributes of those types. The attribute *ident* and the association *accounts* are translated as constants. Each UML-B context gives rise to an Event-B context (i.e. the UML-B tool generates a corresponding Event-B context). Figure 2 also shows the automatically generated Event-B for this example. Each Event-B statement is preceded by its label, which defines its purpose. For example, *ident.type* is a label for the Event-B statement $ident \in CUSTOMER \rightarrow \mathbb{N}$ that defines the type of the *ident* attribute.

Another kind of relationship between two classtypes, super-type (inverse subtype), is used to indicate that the source classtype is more specific than (a subset of) the target classtype. Figure 3 shows an example of subtyping *Account* into *CurrentAccount* and *SavingAccount*. In the generated Event-B machine, the super-type relationship will lead to the type of the classtypes *CurrentAccount* and *SavingAccount* being subsets of the instances of *Account*, i.e. $CurrentAccount \subseteq Account$ and $SavingAccount \subseteq Account$. The subclasstypes *CurrentAccount* and *SavingAccount* give rise to constants in the generated Event-B context.

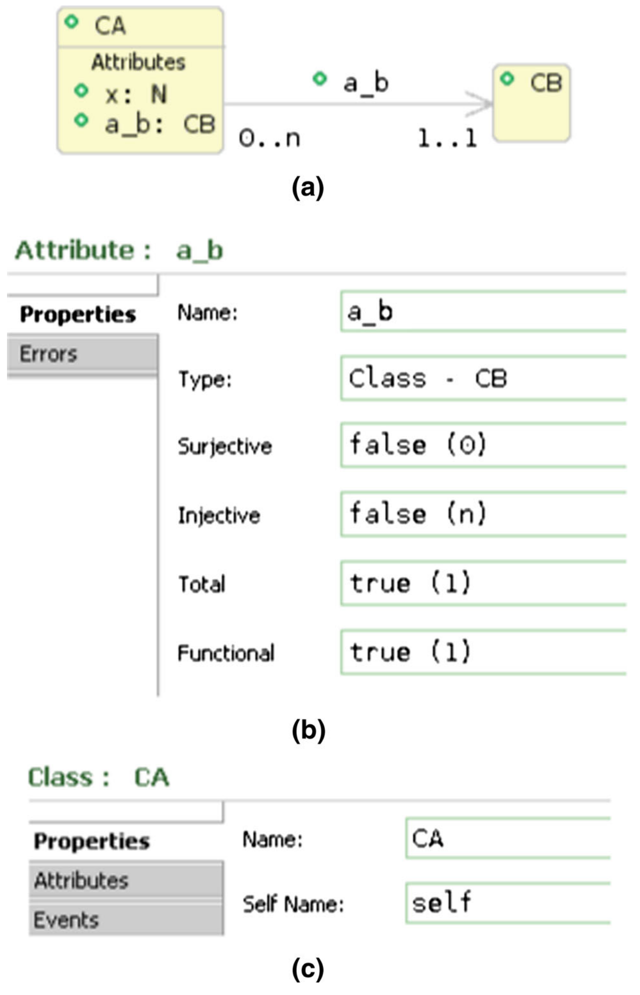
[1] A theorem is a predicate that must be proved from the preceding predicates.

2.3 Class diagram

A class diagram is used to describe the dynamic part of a model. A class diagram contains classes which represent fixed or variable subsets of the classtypes in a context diagram. Classtypes define the immutable fields (including associations), while classes define mutable fields. An object of a class can have both immutable and mutable fields since the class may be a subset of classtype. Like classtypes, each class may have attributes and associations. In addition, each class may own events and state machines. Class events and state machine transitions own actions that may modify the attributes and associations of any class. From each UML-B machine, an Event-B machine is generated. The generated Event-B consists of variables generated from the classes, attributes, associations and state machines and events generated from the class events and state machine transitions. From each UML-B, we also generate an Event-B context to contain the sets and constants that are needed to define the classes and state machines.

Similar to the super-type/subtype relationship between classtypes, a class may also subtype another class giving rise to similar subsets of instances, but in this case, the classes are represented by variables in the generated Event-B machine.

An example of a class diagram is given in Fig. 4a, which consists of classes *CA* and *CB*. In the generated Event-B implicit context, these classes give rise to the sets *CA_SET* and *CB_SET*, which are used as the base types (given, or carrier, sets) for the corresponding classes. (Note that a class may also subtype a classtype in which case these implicit base types are suppressed.) In the generated Event-B machine, the classes give rise to variables. The class *CA* contains an



Attribute : a_b

Properties
Errors

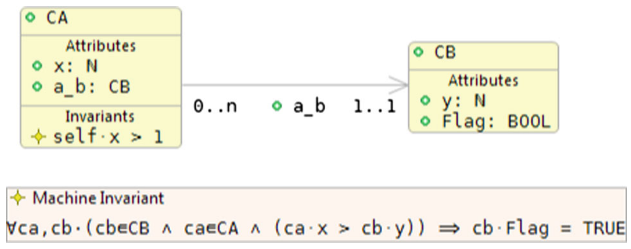
Name:	a_b
Type:	Class - CB
Surjective	false (0)
Injective	false (n)
Total	true (1)
Functional	true (1)

Class : CA

Properties
Attributes
Events

Name:	CA
Self Name:	self

Fig. 4 Example of class diagram and class properties



```
∀ca,cb·(cb∈CB ∧ ca∈CA ∧ (ca·x > cb·y)) ⇒ cb·Flag = TRUE
```

Fig. 5 Example of class and machine invariants

attribute x of type $\mathbb{N}$ and an association a_b of type CB. The multiplicity property for the association a_b shown in Fig. 4b specifies a many-to-one relationship (i.e. total function). A full explanation of association multiplicity may be found in [36]. In the generated Event-B machine, the attributes x and a_b give rise to variables. For each class, attribute and association, a type invariant will be generated. For example, the class CA is typed by an invariant, which specifies that CA is a subset of CA_SET ($CA \in \mathbb{P}(CA\_SET)$) and attribute x is typed by the invariant $x \in CA \rightarrow \mathbb{N}$.

Each class has a self-name property with the default value *self*, i.e. the identifier that is used to represent a contextual instance of a class in class events and invariants. This default may be overridden by the modeller by setting the self-name property. The self-name property of the class CA is shown in Fig. 4c. A class may have events whose parameters, guards and actions are defined explicitly as textual properties. Textual properties of a class, such as invariants, guards and actions, are written in the μB (micro-B) notation [36]. μB is identical to Event-B notation except that it uses an object-oriented style dot notation to navigate ownership of class entities such as attributes and associations. For example, $i.x$ refers to the value of the variable x, which belongs to instance i.

A class may own invariants to express properties of its instances. For example, in Fig. 5, CA has an invariant $self.x > 1$, which says that all instances of CA must have a positive x value. The use of the class's self-name implies universal quantification over the class instances. We may also place invariants in classes when they are not universally quantified over its instances, e.g. card(CA) > 0 in order to indicate their relevance to the class. We may optionally form machine invariants, which are not owned by any class. This is useful to express properties that are not applicable to any class or are equally applicable to several classes. For example, as shown in Fig. 5, the machine invariant relates the class attributes x of CA and y and $Flag$ of CB.

2.4 State machine diagram

Figure 6a shows an example of a state machine SM. The state machine is owned by the class CA. Figure 6b shows its two states, A and B, and the transitions $t1$, $t2$, $t3$ and $t4$. The solid circle is the initial state, and the solid circle with an outer circle is the final state. The translation to Event-B for a state machine can be either a state set representation or state function representation. UML-B allows modellers to switch between these two representations. For the state set representation, each state is represented by a variable which is a disjoint subset of the class instances, CA:

$$A \in \mathbb{P}(CA)$$
$$B \in \mathbb{P}(CA)$$
$$A \cap B = \varnothing$$

That is, variable A represents the set of instances of CA that are in the state A and similarly for B. For a state function representation, a single variable SM represents the state machine as a function mapping each instance of CA to an enumeration of the set of states SM_STATES as follows:

$$SM\_STATES = \{A, B\}$$
$$SM \in CA \longrightarrow SM\_STATES$$

In this paper, the translation to Event-B is described using the state set representation. The generated Event-B machine for *M1* is shown in the Rodin screenshot, in Fig. 7. Each Event-B statement is preceded by its label, which describes its purpose. For example, *CA.type* is a label for the Event-B statement $CA \in \mathbb{P}(CA\_SET)$. The states *A* and *B* of state machine *SM* are represented by variable subsets of *CA* which are disjoint (i.e. $A \cap B = \varnothing$). An instance of *CA* changes its state when a transition fires. For each transition, there is a guard that specifies that the class instance is in the source state (labelled as *.._isin_..*) and actions that specify its entry to the target state (labelled as *.._enterState_..*) and its departure from the source state (labelled as *.._leaveState_..*). The parameter *self* is generated to refer to a non-deterministically selected contextual instance of the class *CA*. A transition from an initial state, such as *t1*, defines a constructor for the class. The translation of *t1* selects an unused instance and adds it to the set of *CA* (labelled *self.type*). A transition to a final state such as *t4* is a destructor which removes an instance from the current instances and from the domain of all the class variables. The transition *t3* is a self-loop transition which does not change state. Hence, in the generated Event-B, the event *t3* has a guard that specifies its source state but no actions to change state.

Invariants and theorems can be attached to classes and/or states and are automatically quantified over the class instances or adorned with an antecedent representing the containing state, as appropriate. A full explanation and examples of these are given in [32].

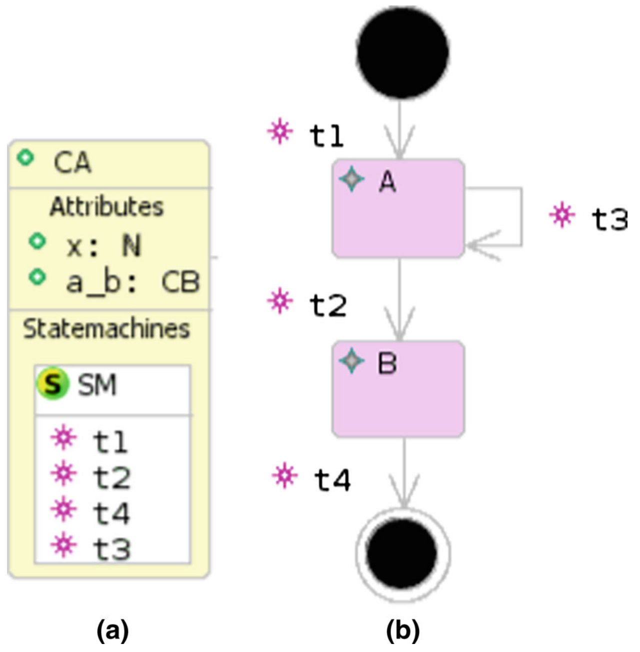
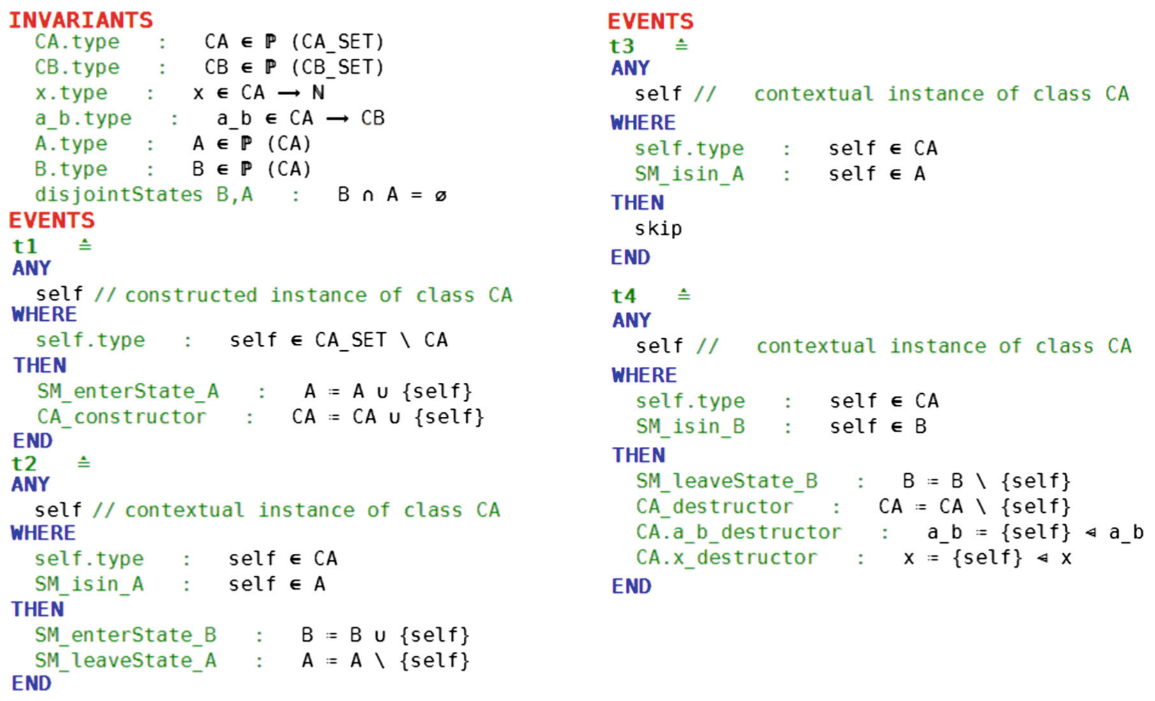


Fig. 6 Example of state machine diagram. **a** Class CA, **b** state machine SM

2.5 Semantics of UML-B

The semantics of UML-B models is given by the generated Event-B model, and Event-B semantics is defined by the cor-



```
INVARIANTS
  CA.type    :   CA ∈ ℙ (CA_SET)
  CB.type    :   CB ∈ ℙ (CB_SET)
  x.type    :   x ∈ CA ⟶ N
  a_b.type    :   a_b ∈ CA ⟶ CB
  A.type    :   A ∈ ℙ (CA)
  B.type    :   B ∈ ℙ (CA)
  disjointStates B,A    :   B ∩ A = ∅
EVENTS
t1   ≙
ANY
  self // constructed instance of class CA
WHERE
  self.type    :   self ∈ CA_SET \ CA
THEN
  SM_enterState_A    :   A ≔ A ∪ {self}
  CA_constructor    :   CA ≔ CA ∪ {self}
END
t2   ≙
ANY
  self // contextual instance of class CA
WHERE
  self.type    :   self ∈ CA
  SM_isin_A    :   self ∈ A
THEN
  SM_enterState_B    :   B ≔ B ∪ {self}
  SM_leaveState_A    :   A ≔ A \ {self}
END
```

```
EVENTS
t3   ≙
ANY
  self //   contextual instance of class CA
WHERE
  self.type    :   self ∈ CA
  SM_isin_A    :   self ∈ A
THEN
  skip
END
t4   ≙
ANY
  self //   contextual instance of class CA
WHERE
  self.type    :   self ∈ CA
  SM_isin_B    :   self ∈ B
THEN
  SM_leaveState_B    :   B ≔ B \ {self}
  CA_destructor    :   CA ≔ CA \ {self}
  CA.a_b_destructor    :   a_b ≔ {self} ⩤ a_b
  CA.x_destructor    :   x ≔ {self} ⩤ x
END
```

Fig. 7 Generated Event-B specification of M1

responding proof obligations. Hence, the semantics of UML-B can be deduced from [32] and [4]. In this section, we give an intuitive semantics which should provide sufficient background for this paper.

Event-B models consist of variables and events. Event-B events are considered to be spontaneous, atomic guarded actions. When the guard of an event is satisfied, the event is enabled and may perform its actions in a single atomic substitution. Several different variables may be altered in parallel during this substitution. An event may have no guards, in which case it is always enabled, or no actions, in which case it does not alter the state of the variables. If one or more events are enabled, one of them will fire. Events do not fire in parallel; if several events are enabled, one of them will be selected non-deterministically to fire and this may change the enabledness of the events. Notice that there is no construct to specify a sequence of events. The feasibility of a sequence of events is only determined by the individual guards and actions of those events. Typically, to specify a sequence of events, a modeller will introduce a variable that resembles a 'program counter' and devise appropriate guards and actions throughout the events to organise them into a sequence. As there is no conditional operator in Event-B, decisions are typically modelled using several alternative events. Within such a group of events, the guards are used to provide the condition for each alternative.

UML-B provides an alternative way to model Event-B variables and events. The constructs in UML-B that define data are classes, attributes, associations and state machines. These data constructs provide additional structure to the types of the variables but in other senses are just Event-B variables. The constructs in UML-B that define events are class events and state machine transitions. Class events are 'lifted' to a set of instances in an object-oriented manner, and transitions impose sequencing (effectively by generating a 'program counter'). Transitions may also be 'lifted' if their state machine is owned by a class. As in Event-B, groups of class events or groups of parallel transitions may be used to represent conditional execution. However, apart from these convenient additions, class events and transitions are just Event-B events. Therefore, like Event-B, UML-B semantics is based on the underlying concept of spontaneous atomic guarded actions that change the state of the variables.

Comparing with other commonly used semantics such as UML, UML-B has no concept of external events that may trigger transitions, no mechanism whereby one transition may invoke another and, as there is no so-called 'big-step', terms such as 'run to completion' have no meaning. All these things can be, and often are, modelled explicitly when required by constructing suitable control variables and guards.

2.6 UML-B model transformation workflow

Despite its name, UML-B is an entirely independent notation from the UML. We suggest that it is 'UML-like' and will feel familiar to UML users; however, it has its own meta-model and no UML models are involved in our discussions here. In [34], we discuss a case study where a UML model of a railway interlocking was translated by hand into UML-B for verification purposes. There are currently no tools that automatically translate UML-B models to or from UML.

Initial versions of UML-B were implemented as extensions to UML using the UML profile mechanism. However, UML is a very rich notation compared with our target language, Event-B, and many features of UML are redundant to our purpose. On the other hand, even where a UML feature seems applicable, Event-B imposes a particular semantics, which is often slightly at odds with that of UML. We found that the combination of unused features and 'false-friends' caused confusion, and hence, we decided to implement UML-B as an independent notation with its own meta-model.

The translation from UML-B model to Event-B is performed programmatically in two steps: firstly by a translation to an internal representation of the Event-B model and then by programmatic generation of the final Event-B model via the Rodin API. Both steps are performed by 'hand-coded' Java programs, and the internal representation of Event-B is represented by 'hand-coded' Java classes. Since this implementation was produced, improvements in meta-model transformation technologies such as QVT [39] and the provision of an EMF framework for Event-B [33] would facilitate an improved model transformation approach.

2.7 UML-B meta-model

The UML-B meta-model [32] defines the abstract syntax of the UML-B language. The UML-B meta-model is described using the *ecore* notation from the Eclipse Modelling Framework (EMF) [37]. The *ecore* notation is based on the OMG's Meta Object Facility (MOF), which is a subset of the UML class diagram notation. Generalisation is used extensively to ensure that common attributes of UML-B model elements are defined in a way that promotes generic, reflective tooling. The meta-model is an exact description of the abstract syntax of the UML-B language and is used to automatically generate repository and editing utility code using the EMF technology. The Eclipse Modelling Framework is a framework and code generation facility for building applications based on a meta-model. Another Eclipse framework, the Graphical Modelling Framework (GMF) [38], is used to automatically generate the code for the UML-B graphical diagram editor.

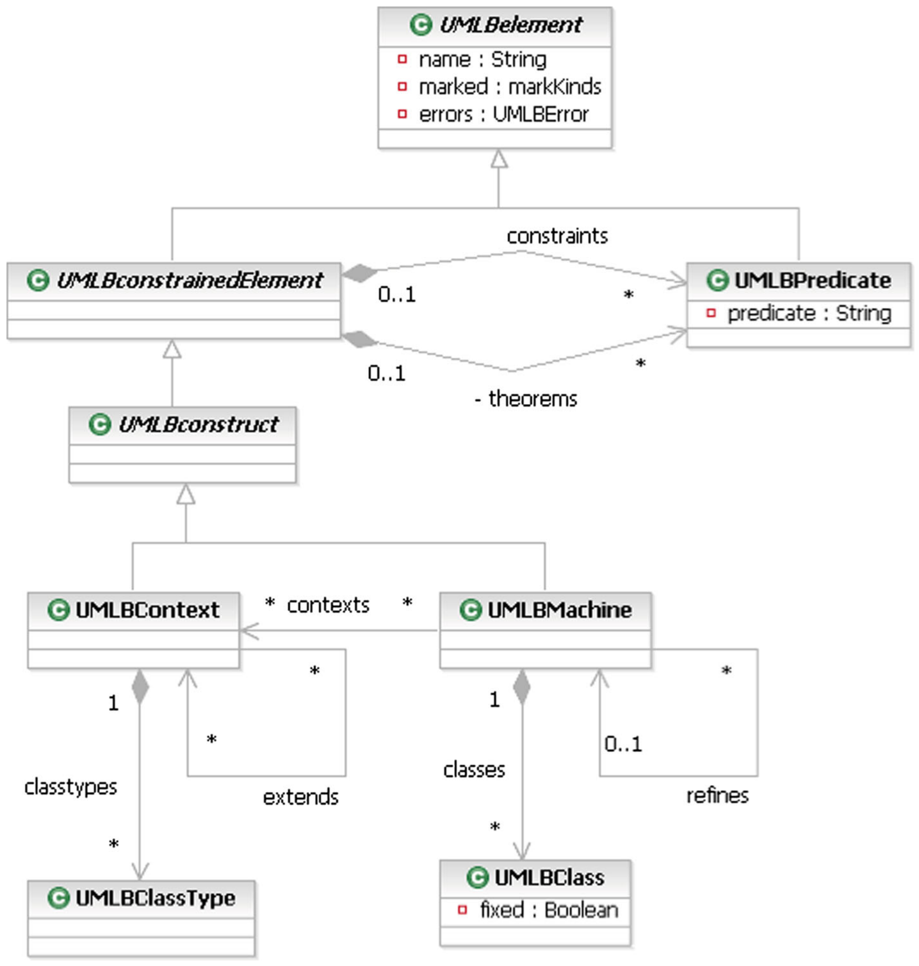


Fig. 8 UML-B meta-model (part of)

To give a flavour of the UML-B meta-model, we show a small part of it in Fig. 8. There are three kinds of relationships used between meta-classes in the meta-model: generalisation, reference and containment. An example of a generalisation (a link with a large triangular arrowhead) is from *UMLBPredicate* to *UMLBelement*, indicating that a *UMLBPredicate* is a specialisation of the meta-class *UMLBelement*. This means that a *UMLBPredicate* can be treated as a *UMLBelement* and includes its properties. An example of a reference (a link with a small arrowhead) is the *refines* link from *UMLBMachine* to itself, which specifies that a machine may refine at most one other machine but may be refined by many machines. An example of a containment (a link with a solid diamond at the source end) is the *classtypes* link from *UMLBContext* to *UMLBClassType*, indicating that a context may contain many classtypes. The meta-class *UMLBelement* is a base meta-class that provides a name property and an error-marking system for recording modelling errors, for all subtyped model elements. *UMLBconstrainedElement*, which subtypes *UMLBelement*, provides a base meta-class for elements that own *constraints* (axioms or invariants) and *theorems* which are elements of another subtype, *UMLBPredicate*. *UMLBPredicate* owns a string property, *predicate*, to contain the text of the predicate. *UMLBMachine* and *UMLBContext* are subtypes of *UMLBConstruct* and hence indirectly *UMLBconstrainedElement* so that they can own such constraints. A *UMLBMachine* can also own a collection *classes* of *UMLBClass*, and a *UMLBContext* can own a collection *classtypes* of *UMLBClassType*. Figure 8 shows a small part of the UML-B meta-model and omits many features such as state machines, variables and events.

3 Overview of refinement in UML-B

We first give an intuitive overview of refinement in UML-B by explaining how it relates to Event-B refinement. Since UML-B is based on the underlying formalism provided by Event-B, so is its notion of refinement. The refinement techniques that are available in UML-B are those of Event-B, but the extra structuring provided by UML-B's higher-level modelling constructs is reflected in its concepts of refinement. We have previously [28] introduced class diagram refinement

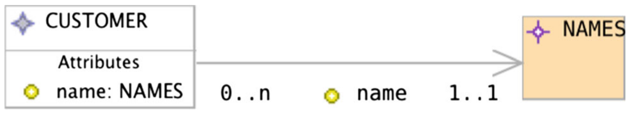


Fig. 9 Example of extended context diagram

and state machine refinement but not context diagram extension. Here, we provide a more complete overview to scope the available features and refinement choices in UML-B.

3.1 Data refinement in UML-B

In Event-B context extension, sets and constants are always retained and may be added to. Hence, in UML-B, classtypes may be extended with new features (attributes, associations, axioms and theorems), but we do not need to specify which old features, or even which old classtypes, are retained; they all are. We need to be able to refer to the old classtypes in order to extend them with new features. In diagrammatic modelling terms, we need a graphic representation of the old classtype as a container for new features. Only the new features represent part of the refined model; the container is a skeleton to provide contextual information that forms part of the definition of the new features. If we do not want to add any new features to a classtype, we can omit the skeleton from the refined model.

For example, referring back to the example in Figs. 1 and 2, in the extended context *CX2*, we might add an association *name* from the extended classtype *CUSTOMER* to a new classtype *NAMES*, as shown in Fig. 9. Notice that we do not need to repeat anything about the previously defined attributes of *CUSTOMER* or the classtype *BANK* since these are still accessible from *CX1*. The only purpose of the extended classtype *CUSTOMER* is to assist in defining the type of the new association.

Variables, on the other hand, may be discarded in Event-B refinement so that they can be refined by new data of a different name and possibly a different type. Refinement relations are captured by specifying invariant properties, called gluing invariants, that relate the corresponding values of new and old variables in the refined model. Variables may be retained by repeating their name in the refinement. Hence, in UML-B refinement, not only do we need refined classes as skeleton containers in order to refine features of the class, but also to indicate that the variable representing the instances of the class is to be retained. In this case, unless we wish to refine the class with some other data variable, we cannot omit the skeleton even if we do not wish to alter its contained features. (UML-B also allows classes to have a fixed set of instances in which case their skeletons are similar to classtypes since their instances are defined as a constant in a context.)

In addition, since class attributes, associations and state machines represent variables, we need to indicate which of these are to be retained. We do this using inherited attributes and refined state machines. The former merely defines that the attribute or association is to be retained, whereas the latter also acts as a container for any nested state machines that are added in the refinement. These retained data features must remain contained in the same classes as their abstract counterparts in order to preserve their types, and refined state machines must contain the same states as their abstract counterparts.

The following schematic illustrates class refinement in terms of attributes.

Class C	**Refined Class C**
a1	a1 *(inherited)*
a2	a3 (*new-data refine* $a2$)
	a4 (*new-superposition*)

In the refinement, class C inherits attribute *a1* and drops attribute *a2*, which is refined by a new attribute *a3*, and C also has a new attribute *a4*. A gluing invariant is needed to relate the dropped attribute to those that replace it. For example, *a2* could be a boolean abstraction of a threshold, which is detailed in *a3*. Hence, the gluing invariant might be

$$\forall c \cdot c \in C \Rightarrow (a3(c) \geq T \Leftrightarrow a2(c) = \mathrm{TRUE})$$

Apart from these considerations, the rules of data variable refinement in UML-B machines are quite flexible in that we may discard any of the previously defined variable data structures (classes, attributes, associations or state machines) and replace them with new ones which may be of a different kind. To do so, we must provide a gluing invariant so that the verifier can establish that the refined events have an equivalent behaviour to the abstract events that they refine.

3.2 Event refinement in UML-B

Event-B events may be refined by retention, renaming or splitting. When refining a retained or renamed event, parameters may be added or replaced, provided that the equivalent of any removed parameter is demonstrated in a witness predicate, guard conjuncts may be added or replaced as long as the overall guard is not weakened, new actions may be added as long as they only modify new variables, and existing actions may be replaced provided they behave in an equivalent way, according to the data refinement. Splitting is a special case where a group of events, representing different conditional cases, all refine the same abstract event that did not reveal the individual cases. New events may be added as long as they only alter new variables. New events are often added as preliminary steps leading up to a refined event.

In UML-B, class events can be refined with equal flexibility. A class event can simply be retained and refined by adding to or replacing its parameters, guards and actions to reflect the data refinements of the class diagram, or the event may be renamed or split into several cases. The requirement to preserve containment observed for class data features does not apply to class events. Since event containment merely defines a parameterisation of the event, we can move events to different classes as long as we provide a witness to demonstrate an equivalent for the lost class instance parameter. The containment of an event in a particular class is chosen for convenience so that a class parameter is automatically generated and guards and actions are automatically lifted to that instance. The same event can always be specified by placing it outside of the class and manually adding the class instance parameter, adjusting the guards and actions accordingly.

The following schematic illustrates the refinement options for five events *e1*, *e2*, *e3*, *e4* and *e5*, which are initially all contained in class *C*.

Refined Class C

e1 (*refines e*1)

e6 (*efines e*2)

e3a (*refines e*3)

e3b (*refines e*3)

Class D

e4 (*refines e*4)

witness : (*selfC* = *selfD.assoc*)

No Class (machine level event)

e5 (*refines e*5)

parameter: ($p \in$ **C**)

witness: (*selfC* = *p*)

The event *e1* remains in class *C* and merely refines its abstract version, whereas *e2* has been renamed *e6*. Event *e3* has been split into two cases *e3a* and *e3b*, which both refine *e3*. In this example, they both remain in class *C*, but it is also possible to move either or both at the same time as splitting. Event *e4* has been moved to another class *D*. In doing so, it loses the implicit *self* parameter (provided by the UML-B to Event-B translation) for the contextual instance of *C*. To satisfy the refinement, we need to provide a replacement and specify its equivalence via a witness. In this example, we assume some association *assoc* from *D* to *C*, which can provide an instance of C. Note that all guards and actions would need to be rewritten to take account of the parameter change. Event *e5* has been moved to machine level (i.e. not contained in any class). Here again, we need to provide a replacement for the lost implicit *self* parameter. In this example, it is provided by a new parameter, which is an instance of class *C*.

The transitions in state machines also represent Event-B events. It is possible to refine UML-B transitions into UML-B events by providing a data refinement that replaces the state machine's states with some other data that provide an equivalent model of state. We might wish to do this when approaching an implementation if the state machine view is considered to be an abstract representation of some concrete system variables. For example, a transition *soundAlarm* with source state *highTemp* could be refined to an event with guard $temperature > limit$ under the data refinement $state = highTemp \Leftrightarrow temperature > limit$.

Usually, however, we do not refine away our state machines but build them up through refinement into more elaborate models of a system where the state machine represents a central 'mode-based' organisation of the system's behaviour. Transition refinement is just as flexible as class event refinement except with respect to its state machine. That is, as with class events, we can retain, rename and split transitions, refining their behaviour to reflect data refinements, but we cannot change the transition's source state since this would not preserve the abstract guard and we cannot change its target state since this would not preserve the abstract behaviour. The only thing we can do to transitions diagrammatically is to split them into two or more cases with the same source and target states. However, we can refine state machines using a particular kind of data refinement where we break down a state into several substates by nesting a new state machine inside the parent state. This allows us to reveal more detailed behaviour in the form of extra transitions (representing new events) between those substates as well as more detailed targets and sources (strengthened guards and refined actions) for the incoming and outgoing transitions, respectively, of the parent state. As a consequence, splitting of transitions in the parent state machine is often necessary when separate cases of the original transition are revealed by the additional detail introduced in the nested state machine.

In summary, a state machine may be refined via two complimentary techniques:

- A transition may be replaced by several transitions representing different subcases of the original transition.
- A state may be elaborated by a nested state machine adding more detailed behaviour.

Figure 10 shows an example where a state *validating* is refined by a nested state machine that adds details concerning PIN number validation. However, this is a manufactured illustration. In the current version of the UML-B tool, nested state machines are modelled in separate diagrams from their parent state and a transition elaboration property is needed to link transitions in a nested state machine to the corresponding incoming and outgoing transitions of the super-state. In a nested state machine, a transition from an initial state elab-

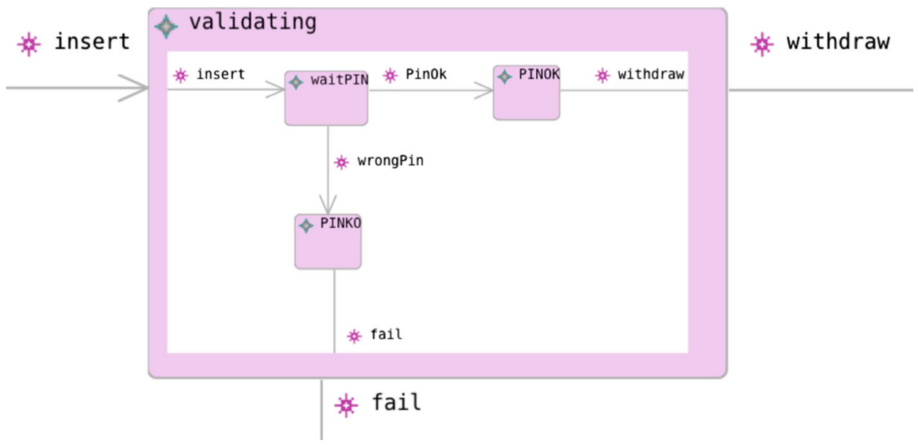


Fig. 10 Example of superposition refinement by state machine nesting

orates exactly one incoming transition to the super-state and a transition to a final state elaborates exactly one outgoing transition from the super-state.

4 Formalisation of rules of refinement in UML-B

In this section, we provide a formalisation of the syntactic rules of refinement in UML-B. We do this using the Event-B notation. We limit ourselves to a description of the aspects that are particular to UML-B and do not cover those features which are a direct reflection of the corresponding Event-B rules. For a formalisation of Event-B refinement, see Abrial's treatment of Event-B [4]. The formalisation is presented in two sections, class diagram refinement and state machine refinement. We introduce a base set *ELEMENT* to represent all UML-B elements and then partition this into subsets to represent the distinct kinds of elements.

$$\text{Partition}(ELEMENT,\ CLASSES,\ ATTRIBUTES,\ EVENTS,\ STATEMACHINES, STATES, TRANSITIONS)$$

We refer to the UML-B machine to be refined as *M1* and the resulting refined machine as *M2*.

4.1 Formal definition of class diagram refinement

We define specific collections, *C1*, *A1*, *E1* and *SM1* of the element types to represent the class diagram of *M1*.

$C1 \subseteq CLASSES$

$A1 \subseteq ATTRIBUTES$

$E1 \subseteq EVENTS$

$SM1 \subseteq STATEMACHINES$

We represent the containment of attributes, events and state machines by their owning classes as functions.

$containmentA1 \in A1 \rightarrow C1$

$containmentE1 \in E1 \rightarrow C1$

$containmentSM1 \in SM1 \rightarrow C1$

We define the components of *M2*, representing the result of refinement of *M1*, in a similar fashion and with corresponding constraints resulting in *C2*, *A2*, *E2*, *SM2*, *containmentA2*, *containmentE2*, *containmentSM2*.

Now that we have defined *M1* and *M2*, we represent the changes made in going from *M1* to *M2*. Let *REM_C1* be the subset of *C1* classes which are removed and *NEW_CL* be the set of new classes added in the refinement.

$REM\_C1 \subseteq C1$

$NEW\_CL \subseteq CLASSES \setminus C1$

Similarly for attributes, events and state machines.

$REM\_A1 \subseteq A1$

$NEW\_AT \subseteq ATTRIBUTES \setminus A1$

$REM\_E1 \subseteq E1$

$NEW\_EV \subseteq EVENTS \setminus E1$

$REM\_SM1 \subseteq SM1$

$NEW\_SM \subseteq STATEMACHINES \setminus SM1$

Let *containmentNEW_ATT* and *containmentNEW_SM* represent the containment of new attributes and new state machines in classes of *M2*, respectively.

$containmentNEW\_ATT \in NEW\_AT \rightarrow C2$

$containmentNEW\_SM \in NEW\_SM \rightarrow C2$

The rules defining the elements of the refined class diagram in machine *M2* are as follows:

Rule C1—Classes of *M2*: The classes of *M2* consist of the classes of *M1* excluding the removed classes and adding the new classes:

$C2 = (C1 \setminus REM\_C1) \cup NEW\_CL$

Rule C2—Attributes of *M2*: The attributes of *M2* consist of the attributes of *M1* excluding the removed attributes and adding the new attributes:

$A2 = (A1 \backslash REM\_A1) \cup NEW\_AT$

Rule C3—Containment of the attributes of *M2*: The containment of attributes in *M2* is the same as that of *M1* omitting the removed attributes and adding the containment of the new attributes:[2]

$$containmentA2 = (REM\_A1 \lhd containmentA1) \\ \cup\ containment\ NEW\_ATT$$

Rule C4—Events of *M2***:** The events of *M2* consist of the events of *M1* excluding the removed events and adding the new events:

$E2 = (E1 \backslash REM\_E1) \cup NEW\_EV$

Rule C5—State machines of *M2***:** The state machines of *M2* consist of the state machines of *M1* excluding the removed state machines and adding the new state machines:

$SM2 = (SM1 \backslash REM\_SM1) \cup NEW\_SM$

Rule C6—Containment of the state machines of *M2*: The containment of state machines in *M2* is the same as that of *M1* omitting the removed state machines and adding the containment of the new state machines:

$$containmentSM2 = (REM\_SM1 \lhd containmentSM1) \\ \cup\ containmentNEW\_SM$$

Notice that we do not introduce any rules about the containment of events in *M2*. This is because events can be freely moved between classes. Instead, we need to define the refinement relationships between the events in *M2* and those in *M1*.

$evRefinementE2 \in E2 \nrightarrow E1$

This is a partial function since some events in *E2* may represent superimposed behaviour and not refine any abstract event (sometimes referred to as 'refining skip'). We also require that every retained event refines itself

$\forall e \cdot e \in E1 \backslash REM\_E1 \Rightarrow evRefinementE2(e) = e$

and every removed event is refined by at least one new event.

$$\forall r \cdot r \in REM\_E1 \\ \Rightarrow \exists n \cdot n \in NEW\_EV \wedge evRefinementE2(n) = r$$

Furthermore, for every event *e* that has moved to a different class, i.e. $containmentE1(e) \neq containmentE2(e)$, we need to add a witness predicate

$W(ca, C2, A2, cc)$

[2] Where $S \lhd r$ removes a set S from the domain of a relation r (domain subtraction).

with,

$ca \in instances(containmentE1(e))$

$cc \in instances(containmentE2(e))$

Utilising the classes *C2* and attributes *A2* of *M2*, the witness predicate *W* establishes a relationship between the disappearing parameter *ca*, representing the contextual class instance of the abstract version of the event, and the new parameter *cc*, representing the contextual class instance of the concrete event.

The final stage is to add sufficient invariants concerning *C1*, *A1*, *C2*, *A2* (and any ancillary variables used) to enable the simulation proofs of refinement. The process of discovering these invariants is discussed in Sect. 7. These steps correspond with Event-B as covered in chapter 5 of [4].

4.2 Formal definition of state machine refinement

We define specific collections to represent the state machine diagrams of *M1*

$S1 \subseteq STATES$

$T1 \subseteq TRANSITIONS$

and represent containment of these states and transitions via functions that map each to its containing state machine.

$containmentS1 \in S1 \rightarrow SM1$

$containmentT1 \in T1 \rightarrow SM1$

We also represent the relationship between the transitions and their source and target states with functions.

$sourceStateT1 \in T1 \rightarrow S1$

$targetStateT1 \in T1 \rightarrow S1$

The source and target of a transition must be contained in the same state machine as the transition.

$$\forall t \cdot t \in T1 \Rightarrow \\ containmentS1(sourceStateT1(t)) = containmentT1(t) \\ \wedge\ containmentS1(targetStateT1(t)) = containmentT1(t)$$

We represent the transition *elaborates* relationship, distinguishing between those that elaborate an outgoing transition of the super-state from those that elaborate an incoming transition.

$elaborateOutgoingT1 \in T1 \rightarrowtail\!\!\!\!\!+ \ T1$

$elaborateIncomingT1 \in T1 \rightarrowtail\!\!\!\!\!+ \ T1$

Note that these functions are injective because each elaborating transition can elaborate at most one parent transition. They are partial because the domains contain both incoming and outgoing elaborating transitions. If an elaborating transition elaborates both an incoming and an outgoing transition, the super-state transition is a self-loop. This is the only case where the domains of the functions intersect.

$$\forall t \cdot t \in dom(\ elaborateOutgoingT1\ ) \cap dom(\ elaborateIncomingT1\ ) \Rightarrow elaborateOutgoingT1\ (t) = elaborateIncomingT1\ (t)$$

We define the components of *M2*, representing the result of refinement of *M1*, in a similar fashion and with corresponding constraints resulting in *S2*, *T2*, *containmentS2*, *containmentT2*, *sourceStateT2*, *targetStateT2*, *elaborateOutgoingT2*, *elaborateIncomingT2*. We define the refinement relationships between the transitions in *T2* and those in *T1*.

$$trRefinementT2 \in T2 \nrightarrow T1$$

This is a partial function since elaborating transitions contribute to other transitions and do not directly represent events. Also, depending on the type of refinement, *T2* may contain new transitions which do not have a refines relationship.

Now that we have defined the components of *M1* and *M2*, we represent the changes made in going from *M1* to *M2*. In UML-B refinement, the structure of a refined state machine is an elaboration of the structure of its abstraction in two possible ways:

1. Transitions may be split into several transitions with the same source and target states.
2. States may be elaborated by a nested state machine.

First, we describe the requirements for (1). For clarity, we describe a refinement where only one transition is refined. In practice, it is possible to split several transitions in one refinement step and add nested state machines at the same time; however, this is equivalent to making a series of simple refinements as described here. Let *tr* be a transition of *T1*, which is to be replaced by a set of new transitions *Ttr* in the refinement.

$$tr \in T1$$
$$Ttr \subseteq TRANSITION \setminus T1$$

Rule T1: States of *M2*. The states of *M2* and their containment are unchanged by this refinement.

$$S2 = S1 \wedge containmentS2 = containmentS1$$

Rule T2: Transitions of *M2*. The transitions of *M2* are the transitions of *M1* with *tr* replaced by *Ttr*.

$$T2 = (T1 \setminus \{tr\}) \cup Ttr$$

Rule T3: Containment of the transitions of *M2*. The containment of transitions in *M2* is the same as that of *M1* except that the new transitions replace *tr* and all have the same container as *tr*.

$$containmentT2 = (\{tr\} \lhd containmentT1) \cup (Ttr \cup \{containmentT1(tr)\})$$

Rule T4: Source and target states of *M2* **transitions**. The source/target states of transitions in *M2* are the same as those of *M1* except that the new transitions replace *tr* and all have the same source/target state as *tr*.

$$sourceStateT2 = (\{tr\} \lhd sourceStateT1) \cup (Ttr \cup (sourceStateT1(tr)\})$$
$$targetStateT2 = (\{tr\} \lhd targetStateT1) \cup (Ttr \cup (targetStateT1(tr)\})$$

Rule T5: Refinement relationship of *M2* **transitions**. In this step, we have not added any new transitions; the 'new' transitions of *Ttr* are actually different cases of *tr*. Therefore, every transition of *Ttr* refines the original transition *tr* and every other non-elaborating transition is unchanged and hence refines itself.

$$dom(trRefinementT2) = T2 \setminus (dom(elaborateOutgoingT2) \cup dom(elaborateIncomingT2))$$
$$\forall t \cdot t \in Ttr \Rightarrow trRefinementT2(t) = tr \forall t \cdot t \in dom(trRefinementT2) \setminus Ttr \Rightarrow trRefinementT2(\ t) = t$$

Next, we describe the requirements for (2) where, in a refinement, a nested state machine is added into a state of *M1*. For clarity, we describe a refinement where only one state is refined. In practice, it is possible to combine this with other refinements including refining several states in one refinement step. However, this is equivalent to making a series of single state refinements.

Let *st* ∈ *S1* be the state which is being refined. Let *IN_Tst* be the set of all incoming transitions into the state *st*, i.e. those transitions of *T1* whose target state is *st*, but whose source state is not.

$$IN\_Tst = \{t | t \in T1 \wedge targetStateT1(t) = st \wedge sourceStateT1(t) \neq st\}$$

Let *OUT_Tst* be the set of all outgoing transitions from the state *st*, i.e. those transitions of *T1* whose source state is *st* and whose target state is not.

$$OUT\_Tst = \{t | t \in T1 \wedge sourceStateT1(t) = st \wedge targetStateT1(t) \neq st\}$$

Let *LOOP_Tst* be the set of all looping transitions from the state *st*, i.e. those transitions of *T1* whose source and target states are both *st*.

$$LOOP\_Tst = \{t | t \in T1 \wedge sourceStateT1(t) = st \wedge targetStateT1(t) = st\}$$

Let *sm* be the new nested state machine to be added to *st* and let *Ssm* and *Tsm* be its new sets of states and transitions, respectively.

$$sm \in STATEMACHINES \setminus SM1$$
$$Ssm \subseteq STATES \setminus S1$$
$$Tsm \subseteq TRANSITIONS \setminus T1$$

Let *sourceStateTsm* and *targetStateTsm* map each new transition to its source and target state, respectively.

$$sourceStateTsm \in Tsm \rightarrow Ssm$$
$$targetStateTsm \in Tsm \rightarrow Ssm$$

The new transitions are partitioned into initial, final, internal elaborating and internal non-elaborating transitions.

$$partition(Tsm, INI\_Tsm, FIN\_Tsm, INT\_ELAB\_Tsm, NON\_ELAB\_Tsm)$$

We now define one-to one mappings (injections) to represent the elaborates relationship of the new state machine. Each initial transition of *sm* elaborates one incoming parent transition of *st*, each final transition elaborates an outgoing parent transition, and each internal elaborating transition elaborates a parent loop transition.

$$elaborateIncomingTsm \in INI\_Tsm \rightarrowtail\!\!\!\!\twoheadrightarrow IN\_Tst$$
$$elaborateOutgoingTsm \in FIN\_Tsm \rightarrowtail\!\!\!\!\twoheadrightarrow OUT\_Tst$$
$$elaborateLoopTsm \in INT\_ELAB\_Tsm \rightarrowtail\!\!\!\!\twoheadrightarrow LOOP\_Tst$$

Rule S1: States of *M2*. *M2* has all the states of *M1* as well as the states of the new state machine *sm*.

$$S2 = S1 \cup Ssm$$

Rule S2: Containment of the states of *M2*. The containment of states in *M2* is the same as that of *M1* but adding the containment of the new states in *sm*.

$$containmentS2 = containmentS1 \cup (Ssm \cup \{sm\})$$

Rule S3: Transitions of *M2*. *M2* has all the transitions of *M1* as well as the transitions of the new state machine *sm*.

$$T2 = T1 \cup Tsm$$

Rule S4: Containment of the transitions of *M2*. The containment of transitions in *M2* is the same as that of *M1* but adding the containment of the new transitions in *sm*.

$$containmentT2 = containmentT1 \cup (Tsm \cup \{sm\})$$

Rule S5: Source states of ***M2*** **transitions**. The source state mapping of the transitions in *M2* is the same as that of *M1* but adding the source state mapping of the new transitions in *sm*.

$$sourceStateT2 = sourceStateT1 \cup sourceStateTsm$$

Rule S6: Target states of ***M2*** **transitions**. The target state mapping of the transitions in *M2* is the same as that of *M1* but adding the target state mapping of the new transitions in *sm*.

$$targetStateT2 = targetStateT1 \cup targetStateTsm$$

Rule S7: Outgoing elaborating transitions of *M2*. The outgoing elaborations of the transitions in *M2* are the same as those of *M1* but adding the outgoing and loop elaborations of the new transitions in *sm*.

$$elaborateOutgoingT2 = elaborateOutgoingT1 \cup elaborateOutgoingTsm \cup elaborateLoopTsm$$

Rule S8: Incoming elaborating transitions of *M2*. The incoming elaborations of the transitions in *M2* is the same as that of *M1* but adding the incoming and loop elaborations of the new transitions in *sm*.

$$elaborateIncomingT2 = elaborateIncomingT1 \cup elaborateIncomingTsm \cup elaborateLoopTsm$$

Rule S9: Refinement relationship of ***M2*** **transitions**. The transitions *Tsm* are new and therefore do not have refinement relationships. Every other non-elaborating transition in *T2* is unchanged and hence refines itself.

$$dom(\ trRefinementT2\ ) = T2 \setminus (Tsm \cup dom(\ elaborateOutgoingT2\ ) \cup dom(\ elaborateIncomingT2\ ))$$
$$\forall t \cdot t \in dom(\ trRefinementT2\ ) \Rightarrow trRefinementT2\ (\ t\ ) = t.$$

5 Enhancement of the UML-B meta-model to support refinement

This section describes enhancements to the UML-B meta-model, which were required to support refinement of UML-B models. We refer to the version of UML-B before the extensions as UML-B Version 1 and after the extension as UML-B version 2. (UML-B version 1 corresponds to reference [32]. UML-B version 2 was extended and used for, but not reported in, reference [28].) Version 1 already contained some features that are needed for refinement. These were a *refines* reference feature from *UMLBMachine* to itself to support machine refinement, a *refines* reference feature from *UMLBguardedAction* to itself to support event and transition refinement relationships (*UMLBguardedAction* being a super-type for both *UMLBEvent* and *UMLBTransition*) and an *extends* reference feature from *UMLBContext* to itself to support context extension. The support for refinement of model elements that represent data was entirely missing from version 1 and required the introduction of new meta-classes to represent data that had already been introduced in previous refinement levels. The support for refinement of model elements that represent events is simpler because this can be achieved by introducing new event elements that reference the abstract ones. Indeed, this is how Event-B manages event refinement. Support for event refinement was already present in version 1 apart from one minor addition that was needed to support refinement of state machine transitions.

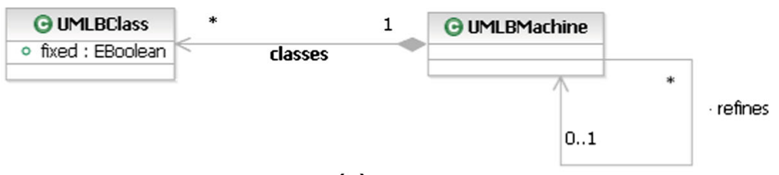
5.1 Support for refinement of class diagrams

In UML-B version 1, there was no mechanism to distinguish a class that was being refined from a newly defined class. Although a class could be retained by repeating it in the refined class diagram, the Event-B generation would produce invariants to redefine the variables representing the class instances at each refinement level, leading to overcomplicated Event-B. Similarly, there was no way to distinguish attributes and associations that were being retained in the refined class from those that were newly introduced. The Event-B generation would reproduce invariants to redefine old attributes and associations at each refinement level. To provide better support for refinement of classes, a new meta-class, *UMLBRefinedClass*, was introduced to represent the 'skeleton' of a refined class (Fig. 11). This, and the meta-class for new classes *UMLBClass*, both subtype a common ancestor *UMLBabstractClass* which provides the type for the containment of classes in a machine. *UMLBRefinedClass* has none of the properties of *UMLBClass*, such as name and super-type, since a refined class should not be able to redefine these properties. When such properties are needed by the refined class, for example to display a label on the class diagram or to generate the Event-B representation of a contained attribute, they must be obtained from the original class that has been refined. Therefore, the only property possessed by the meta-class *UMLBRefinedClass* is a *refines* reference. The target of this reference is of type *UMLBabstractClass* to allow for a chain of several refinement levels. That is, the target of the *refines* reference may, itself, be a refined class.

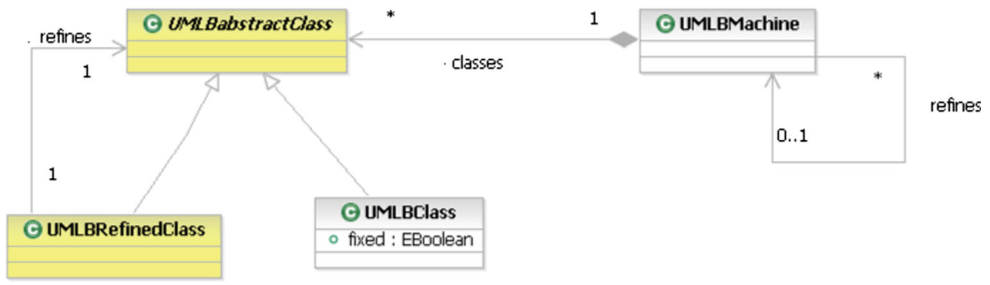
A similar arrangement was also introduced to support attributes that are to be retained in a refined class. In this case, we refer to them as 'inherited' attributes since they cannot be refined in any sense. The new meta-classes are *UMLBabstractAttribute* and *UMLBInheritedAttribute*, the latter having a reference *inherits* targeting the former and the former being the target type for the containment of attributes in classes. Note that UML-B associations are an alternative concrete syntax for UML-B attributes; hence, no separate meta-model arrangement is required for inherited associations.

5.2 Support for extension of context diagrams

In UML-B version 1, there was no mechanism to distinguish a classtype that was being extended with new features from a newly defined classtype. If the modeller repeated the classtype in order to extend it, the Event-B generation would produce constants and axioms to redefine the classtype in the context extension, leading to Event-B errors. A pattern of meta-classes similar to that used for classes and attributes was introduced to support extension of classtypes. The new meta-classes are *UMLBabstractClassType* and *UMLBExtended-*



(a)



(b)

Fig. 11 UML-B meta-model enhancements to support the refinement of classes. **a** Previous meta-model, **b** changes to the previous meta-model

ClassType, the latter having a reference *extends* targeting the former and the former being the target type for the containment of classtypes in contexts. Note that nothing is needed for classtype attributes and classtype associations because they are always visible through extensions and therefore do not need to be retained in an extension.

5.3 Support for refinement of state machine diagrams

In UML-B version 1, there was no mechanism to distinguish a state machine that was being refined from a newly defined state machine. If the modeller repeated the state machine in order to refine it, the Event-B generation would repeat data and constraints to represent the state machine state in the new refinement level, leading to Event-B errors. A pattern of meta-classes similar to that used for classes and attributes was applied both for the refinement of state machines and for the refinement of states, resulting in the new meta-classes *UMLBabstractStatemachine*, *UMLBRefined-Statemachine*, *UMLBabstractState* and *UMLBRefinedState*. Some features of state machines, such as name, are owned by *UMLBStatemachine* and therefore not available to *UMLBRefinedStatemachine* and must be obtained via the refines relationship, whereas some features of state machines are owned by *UMLBabstractStatemachine* so that they are also available for redefinition in a refined state machine. Most notably, a refined state machine owns its own set of transitions so that transition refinement can be undertaken. The containment of states is also moved to *UMLBabstractStatemachine*; however, unlike transitions, this is a consequence of the need to create new refined states at each refinement level rather than a modelling facility.

At this point, we note that we have used the same meta-model pattern for all of the UML-B elements that represent Event-B data (class, classtype, attribute, state machine and state). The generic pattern may be described as the containment of instances of an abstract super-type which are partitioned into actual model elements and 'skeleton' model elements, the latter having a reference relationship to an instance of the abstract super-type which should eventually, via transitive closure, provide an actual model element.

There is one final enhancement to the UML-B meta-model which was introduced in UML-B version 2. It concerns the relationships between the newly introduced transitions of a nested state machine and those incoming and outgoing transitions connected to its parent state. The nested state machine contains some initial and final transitions which contribute to existing events that have, in the previous level, already been generated by the incoming and outgoing transitions of the parent state. For these initial and final transitions, we need to specify which parent transition they contribute to so that the generation can add their guards and actions to the corresponding existing event. Similarly, internal transitions may contribute to self-looping transitions of the parent state. To provide the reference between the nested transition and the parent transition, two new references, *elaborates* and its inverse *isElaboratedBy*, were added to the meta-model as a self-reference on the meta-class *UMLBTransition*.

6 Overview of ATM case study in UML-B

This section presents the development of an ATM case study in UML-B using refinement. The development uses all the extensions of UML-B meta-model.

The package diagram in Fig. 12 shows the contexts, the five levels of machines and their relationships where a machine sees a context, a context extends another context, and a machine refines another machine. The summary of the five machine levels is given here.

Abstract machine (ATM_A): The abstract machine models the accounts in a bank and a number of operations that may be performed on the accounts.

First Refinement (ATM_R1): The first refinement introduces a set of ATMs as a medium to withdraw money or to check an account balance.

Second Refinement (ATM_R2): The second refinement introduces a concept of PIN number and models an explicit validation for cards.

Third Refinement (ATM_R3): The third refinement introduces the request and response communication between an ATM and the bank and splits a withdrawal into a bank transition and an ATM transition.

Fourth Refinement (ATM_R4): The fourth refinement models the send and receive events of the request and

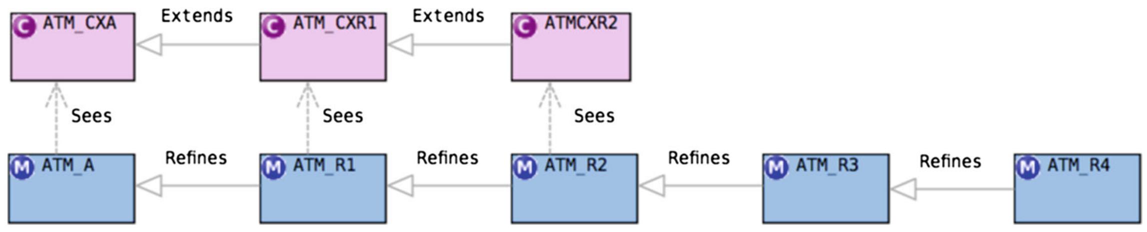


Fig. 12 ATM package diagram

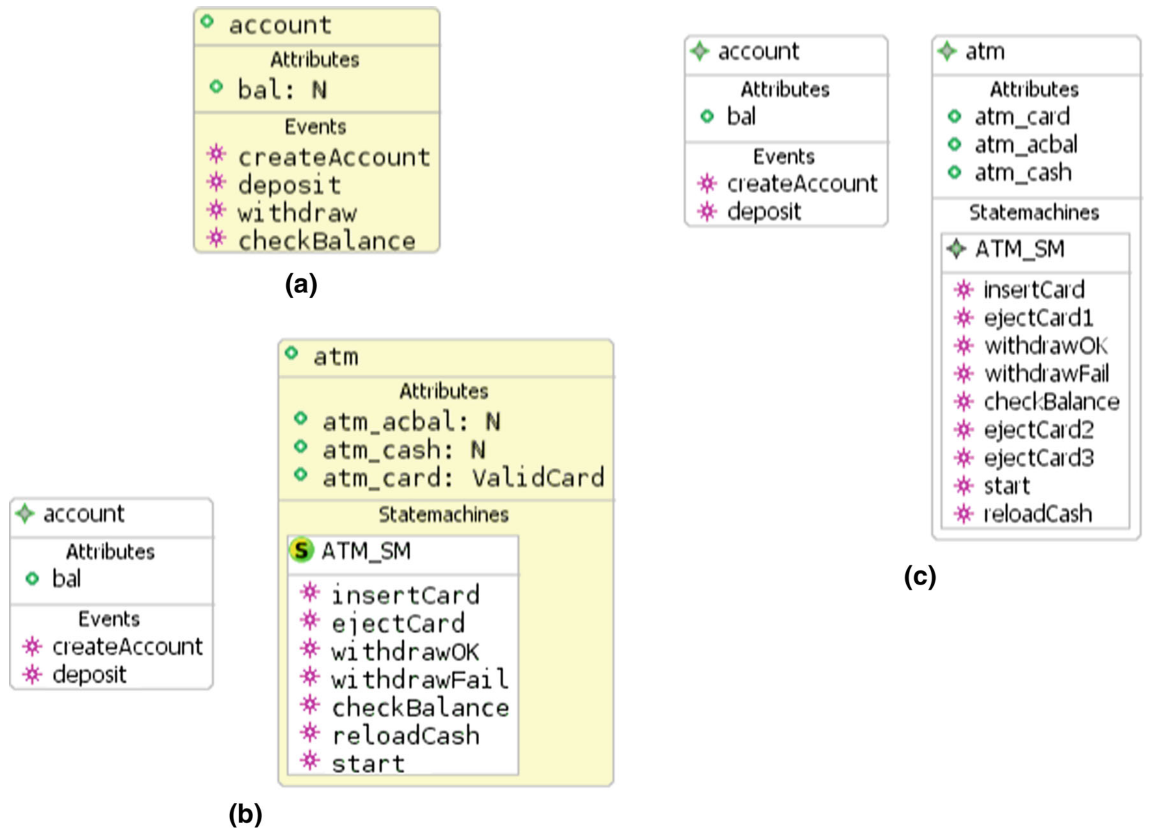


Fig. 13 Class diagrams of ATM. **a** Class diagrams of ATM_A, **b** class diagrams of ATM_R1 and **c** class diagrams of ATM_R2

response communication between ATMs and the bank. This is done by adding a receive event for each request and adding a send event for each response. The send event for request refines the abstract request event. The receive event for response refines the abstract response event. The fourth refinement also introduces a set of requesting ATMs whose requests are being processed by the bank.

We outline some class and state machine diagrams of the ATM case study referring to the refinement rules in Sect. 4. Details of the development in UML-B can be found in [28]. Figure 13 shows some of the class diagrams of the ATM case study. The machine *ATM_A* consists of a class *account* (Fig. 13a) with its attribute *bal* and four events, namely *createAccount*, *deposit*, *withdraw* and *checkBalance*. The *account* class represents the set of accounts that currently exist in the system. The attribute *bal* represents the balance of an account.

The class diagram of *ATM_R1* (Fig. 13b) contains the new class *atm* and a refined class *account* that refines the *account* class of *ATM_A*. These two element of classes in *ATM_R1* referred to refinement *Rule C1*. The class *atm* has three attributes which are *atm_acbal*, *atm_cash* and *atm_card*. The attribute *atm_acbal* represents an account balance after each cash withdrawal or checking of balance transaction via an ATM. The attribute *atm_cash* represents a stock of cash in an ATM. The attribute *atm_card* represents a card in an ATM. The refined class *account* inherits the *bal* attribute. These refinement elements refer to *Rule C2* and *C3*. The refined class *account* refines the two events, namely *createAccount* and *deposit* of the abstract *account* class of machine *ATM_A*. The other two events of its abstract class, namely *withdraw* and *checkBalance*, are moved to the new class *atm* in this refinement level as transitions in the state machine *ATM_SM* of the class *atm*. At the abstract level, we specify the effect of a withdrawal on the account balance. In the refinement, we further specify that the withdrawal takes place via an ATM. At the abstract level, it is natural to specify the withdrawal as an event of the *account* class, while in the refinement, it is natural to specify it as a transition of the *atm* class. The events element refers to *Rule C4*, where *withdraw* and *checkBalance* events are removed and no new event is added. The refinement rules referred are *Rule C5* and *Rule C6* when adding the state machine *ATM_SM*. The transitions of the state machine are explained later in this section.

The class diagram (Fig. 13c) of *ATM_R2* contains the two refined classes that refine the *account* and *atm* classes of *ATM_R1* machine. The refined class *atm* of *ATM_R2* contains the refined state machine *ATM_SM*.

Figure 14 shows some state machine diagrams of the ATM case study. The state machine *ATM_SM* in Fig. 14a parti-

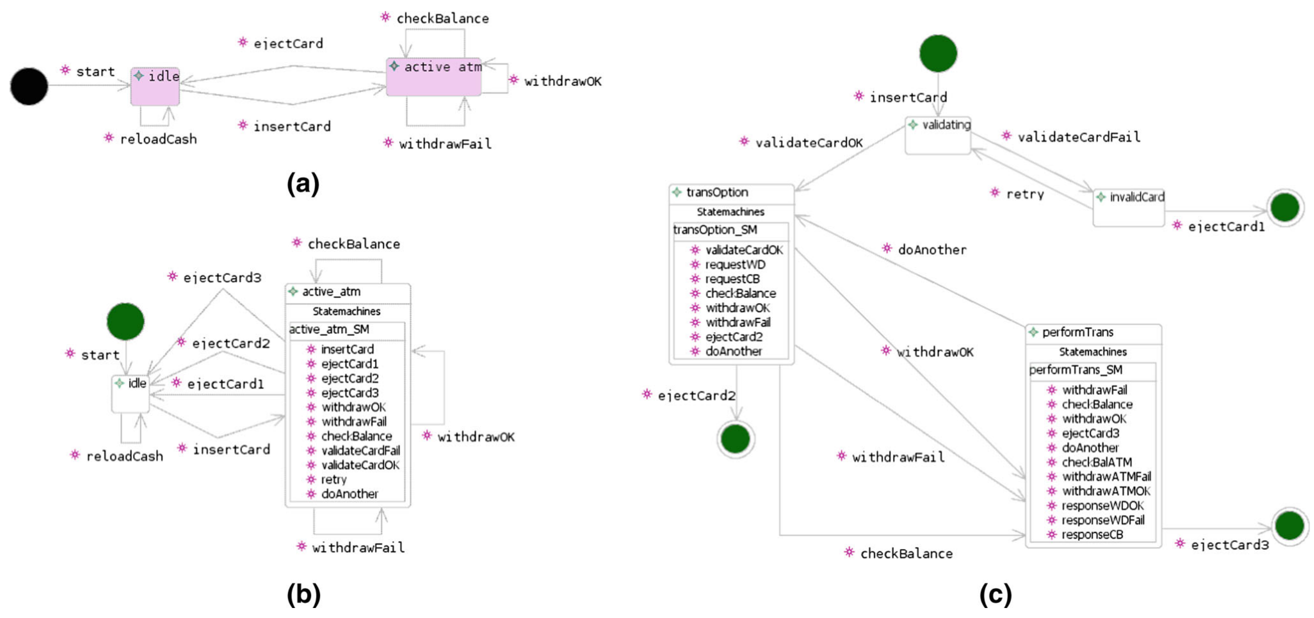


Fig. 14 State machine diagrams. **a** State machine ATM_SM of class ATM of machine AT_R1, **b** refined state machine ATM_SM of refined class ATM of machine ATM_R2 and **c** refined state machine active_atm_SM of ATM_R3

tions the behaviour of an ATM into either an *idle* state (i.e. not being used/not active) or *active_atm* state (i.e. is being used). If a transition *t1* is triggered and the current state is the source state of *t1*, the ATM changes state. The transition *start* creates an instance of ATM and adds it to the set *atm_card*, initialises its stock of cash as *MAX_CASH* and changes its state to *idle*. The *insertCard* transition can be triggered when an ATM is in the *idle* state and the inserted card is a valid ATM card. When it is triggered, it changes the ATM state from *idle* to *active_atm*. The *reloadCash* transition can trigger when an ATM is in the *idle* state and the ATM cash amount is less than the *MAX_CASH*. The *reloadCash* transition will top up the ATM cash to the maximum amount *MAX_CASH*. The *ejectCard* transition changes an ATM state from *active_atm* to *idle* and removes the ATM from the set *atm_card*. While an ATM is in *active_atm* state, it means that an ATM user can use it for withdrawal or checking an account balance (i.e. *checkBalance* transition). The *withdrawOK* transition represents a successful withdrawal transaction, whereas the *withdrawFail* transition represents a failure possibly because the withdrawal amount exceeds the account balance.

The refined class *atm* of *ATM_R2* contains the refined state machine *ATM_SM* (Fig. 14b), which contains the two refined states that refine the states *idle* and *active_atm* of the state machine *ATM_SM* of *ATM_R1*. The transition *ejectCard* is split into three transitions, namely *ejectCard1*, *ejectCard2* and *ejectCard3*, which refine *ejectCard*. The other five transitions refine themselves. This refinement refers to *Rules T1*, *T2*, *T3*, *T4* and *T5*. These rules assumed that the state *active_atm* does not have state machine *active_atm_SM* yet. For *Rule T1*, the states of *ATM_R2* and their containment are the same as *ATM_R1*. Referring *Rule T2*, *ejectCard1*, *ejectCard2* and *ejectCard3* replaced *ejectCard*. As in *Rule T3*, the container of the replacing transitions is the same as *ejectCard*'s, similarly for their source and target states as in to *Rule T4*. The refinement relationship refers *Rule T5*.

The state machine *active_atm_SM* of *ATM_R2* is like the one in Fig. 14c but without nested state machines in states *transOption* and *performTrans*. When the state machine *active_atm_SM* is added, referring to *Rule S1*, the states of *ATM_R2* are extended to include new states *validating*, *invalidCard*, *transOption* and *performTrans*. *Rule S2* defined the containment of all the states. The new states contained in the state machine *active_atm_SM*. For *Rule S3*, new transitions consist of initial elaborating, final elaborating, internal elaborating and internal non-elaborating. Initial elaborating is *insertCard*. Final elaborating transitions are *ejectCard1*, *ejectCard2* and *ejectCard3*. Internal elaborating transitions are *withdrawOK*, *withdrawFail* and *checkBalance*. Internal non-elaborating transitions are *validateCardOK*, *validateCardFail*, *retry* and *doAnother*. As in *Rule S4*, the container of new transitions is the state machine *active_atm_SM*. As in *Rule S5* and *Rule S6*, all new transitions must have their source and target states. Referring *Rule S7*, the outgoing elaborate relationships include the new final elaborating and internal elaborating transitions with respective outgoing transition of super-state *active_atm*. For *Rule S8*, the incoming elaborate relationships include the new initial final elaborating and internal elaborating transitions with respective incoming transition of *active_atm*. Referring *Rule S9*, all non-elaborating old transitions refine themselves.

Figure 14c shows a refined state machine *active_atm_SM* of machine *ATM_R3*, which shows that the refined state *transOption* and the refined state *performTrans* have nested state machines. This approach of elaborating states with substates in refinement supports an incremental refinement approach. The ATM case study has shown that the extensions of the meta-model and drawing tools are working as expected.

The ATM case study reported in this paper differs slightly from the one presented in [28] although they are based on the same version of the UML-B meta-model. The differences are as follows:

- In [28], we did not have classtypes of *ATM*, *Card* and *Pin* to give rise to the sets *ATM*, *Card* and *Pin* in an Event-B context. Instead, the sets are generated from the classes in the class diagram and contained in the Event-B implicit context. Thus, there were no context diagrams to be extended with new classtypes. In the case study, we wanted the ability to extend the classtypes in refinements to introduce immutable attributes. Therefore, for the ATM model in this paper, we created a UML-B context diagram containing the three classtypes, which were then used as instances for the classes.
- The refinement strategy is slightly changed in this paper. In [28], the splitting of *withdrawal* into bank and ATM transitions is done in the second refinement. We think this is not reasonable because the transition withdrawal request that causes the withdrawal ATM transition is not introduced until the third refinement. It is more reasonable to delay introducing the withdrawal ATM transition until the third refinement. This improves the sequence of transitions between request and response. Ideally, the response should come when there exists a request. In this case, *requestWD* is the request made by the user via an ATM machine, while *responseWDOK* and *responseWDFail* are the responses to the user from the ATM. This improves the cohesiveness of the refinements and allows the second refinement to deal with pin validation.
- In this paper, we have formalised the refinement rules and explicitly refer to them in the ATM case study.

An archive of the UML-B development of the ATM case study can be downloaded.[3] UML-B is a plug-in to the Rodin platform, which can be downloaded.[4] UML-B can then be installed from the update site contained in Rodin (Help-Install New Software: select 'Rodin' update site). Instructions on using Rodin (including installation of plug-ins) are available.[5]

[3] ATM case study: http://eprints.soton.ac.uk/346101/.

[4] http://sourceforge.net/projects/rodin-b-sharp/files/Core_Rodin_Platform/.

[5] http://handbook.event-b.org/current/html/tut_install_plugins.html.

Table 1 Statistics from the proof effort

Machines	POs	aPOs	iPOs
ATM_A	4	4	0
ATM_R1	47	47	0
ATM_R2	68	68	0
ATM_R3	167	160	7
ATM_R4	149	142	7
Total	435	421	14

7 Proofs and invariants

This section discusses the proofs of the UML-B case study and also the construction of gluing invariants using Rodin provers.

All the proof obligations (POs) for the five machines of the ATM case study were generated and proved using the Rodin tool provers [3]. The statistics are outlined in Table 1 showing the total POs for each level (POs), the number of POs which are automatically discharged (aPOs) and the number of POs which are interactively discharged (iPOs).

In *ATM_R3*, there are seven interactively discharged POs. Three POs are discharged manually by proving that two related states are disjoint and another four are proved by rewriting the partition invariant into its definition. A similar way is used to prove the seven interactively POs in *ATM_R4*. Two POs are discharged by manually proving that two states are disjoint and the other five POs are discharged by rewriting the partition invariant.

Where refinements have been made a gluing invariant may be needed to relate the abstract data to the new data. In general, finding suitable invariants can be non-trivial and care must be taken not to introduce unnecessary invariants which can increase the proof burden as every event must be shown to preserve them. We discuss two alternative approaches to constructing gluing invariants: firstly by discovery from information provided by the prover and secondly by design, from information in the model. Both methods are based on the refinement pattern of nesting state machines but are not in any way specific to the ATM case study.

7.1 Constructing a gluing invariant by discovery

Some of the gluing invariants are constructed by using guidance from the undischarged proof obligations. We describe first our method of discovering the gluing invariants. Then, we give some examples of discovering the invariants for the ATM case study.

In this paragraph, we describe our method for discovering the gluing invariants. We inspect an undischarged PO, $H \vdash G$, (consisting of some available hypotheses H and a goal G) and construct an invariant of the form $\forall x\cdot H' \Rightarrow G$ where *H'* is a subset of the list of hypotheses *H* and *x* represents the list of free variables that correspond to event parameters. The selection of hypotheses *h* from *H* to appear in *H'* is based on these rules:

1. *h* is of the form $p \in S$, where *p* is an event parameter and *S* represents a state of a state machine. In particular, *S* is the substate of a nested state machine.
2. The free variables of *h* are included in the free variables of *G*.

In the next paragraphs, we describe some examples of discovering invariants using the above rules (1) and (2). One of the discovered gluing invariants is in the third refinement (*ATM_R3*). An attempt to construct the invariant is done by using the interactive prover. The *ATM_R3* was run in a proving perspective without having any gluing invariant, which results in a number of undischarged proof obligations. The first undischarged PO is given here as an example. The prover cannot discharge the guard $atm\_cash(selfATM) \geq am$ of the event *withdrawOK*. The hypotheses and the goal are as follows:

Hypotheses:

$am \in \mathbb{N}$

$ac \in account$

$selfATM \in atm$

$c \in ValidCard$

$selfATM \in reqWD$

$selfATM \in dom(atm\_card)$

$atm\_card(selfATM) = c$

$bal\ (\ ac\ ) \geq am$

$card\_account\ (\ c\ )) = ac$

$selfATM \in dom\ (\ atm\_wdam\ )$

The goal:

$atm\_cash\ (\ selfATM\ ) \geq atm\_wdam$
$(\ selfATM\ )$

From the above PO, the prover is trying to prove that the cash in an ATM is greater or equal to a given withdrawal amount. This is true for any successful cash withdrawal. According to rule (1), *selfATM* is the event parameter concerned in the goal and *reqWD* is a substate of a nested state machine *performTrans_SM* (Fig. 14c). Therefore, from the list of hypotheses, $selfATM \in reqWD$ is selected as one of the hypotheses in the gluing invariant. Also, *atm_wdam* is the free variable included in the goal. Thus, according to rule (2), $selfATM \in dom(atm\_wdam)$ is also selected as the hypotheses in the gluing invariant. The required invariant is represented in Event-B as follows:

$\forall selfATM \cdot selfATM \in reqWD$
$\wedge selfATM \in dom\ (\ atm\_wdam\ )$
$\Rightarrow atm\_cash\ (\ selfATM\ ) \geq$
$atm\_wdam\ (\ selfATM\ )$

Another example is the gluing invariant in the fourth refinement (*ATM_R4*). The prover cannot discharge the guard $selfATM \in dom(atm\_card)$ of the event *withdrawOK*. The hypotheses and the goal are as follows:

Hypotheses:

$selfATM \in atm$

$selfATM \in recvdReqWD$

$selfATM \in atmB$

$selfATM \in dom(atm\_wdamB)$

$atm\_cardB(selfATM) = c$

$am = atm\_wdamB(selfATM)$

$card\_account(atm\_cardB$
$(selfATM)) = ac$

The goal:

$selfATM \in dom(atm\_card)$

The prover is trying to prove that a given ATM has an ATM card in it. Similar to the first example, following rule (1), *selfATM* is the event parameter concerns in the goal and *recvdReqWD* is the substate of the nested state machine *reqWD_SM* of the substate *reqWD*. Thus, $selfATM \in recvdReqWD$ is selected forming the gluing invariant. The required invariant is represented in Event-B as follows:

$\forall selfATM \cdot selfATM \in recvdReqWD$
$\Rightarrow selfATM \in dom(atm\_card)$

The task of finding gluing invariant is also the same for the undischarged PO involving the guard $selfATM \in dom(atm\_card)$ of the event *checkBalance*, i.e. when *selfATM* is in the substate *recvdReqCB*. The two invariants can be combined forming a single invariant as follows:

$\forall selfATM \cdot selfATM \in (recvdReqWD$
$\cup recvdReqCB) \Rightarrow selfATM \in dom(atm\_card)$

Another example of finding the gluing invariant in the fourth refinement is when the prover cannot discharge the guard $atm\_card(selfATM) = c$ of the event *withdrawOK*. The hypotheses are the same as the first example of *ATM_R4*, and the goal is as follows:

$atm\_card(selfATM) = atm\_cardB(selfATM)$

Similarly, from the PO, using rules (1) and (2), the discovered gluing invariant is as follows:

$$\begin{aligned}&\forall\, selfATM \cdot selfATM \in recvdReqWD\\ &\quad \wedge\ selfATM \in dom(atm\_cardB)\\ &\quad \Rightarrow atm\_card(selfATM) = atm\_cardB\end{aligned}$$

However, these rules are heuristics and they do not provide a complete method for verifying refinements. But they were sufficient to prove the refinements in our ATM development.

We would like to point out that UML-B is not a purely graphical notation. In particular, we need to use a textual representation of gluing invariants in order to prove the refinement. All the discovered invariants are specified in UML-B as invariants in class diagrams.

7.2 Constructing a gluing invariant by design

While it is attractive to let the prover indicate the invariants it needs, and to have a heuristic for constructing them mechanistically, it is also possible to construct sufficient gluing invariants purposefully. This can be done either before running the prover, or using the proof obligation goal as a hint. The latter differs from the invariant discovery method in that the gluing invariant is constructed by examining the model rather than examining the goal and hypotheses; the goal is only used as a hint of what to look for in the model.

An abstract model has many possible valid refinements, but when modelling, we choose one particular refinement and wish to verify that it is correct. The prover can verify that it is a refinement, but it cannot tell whether it is the particular refinement we intended unless we provide some extra information. By expressing the linkages between the abstract model and the refined one, a gluing invariant indicates 'why' it is a refinement and hence indicates 'which' refinement was intended. If we let the prover tell us which gluing invariant to use, we lose this extra verification condition and there is a danger that we will end up with a verified but wrong system. In practice, the likelihood of constructing a valid but wrong refinement may be remote, particularly as we get the opportunity to examine the gluing invariants thrown up by the prover since the discovery method is not fully automatic. However, there is some motivation at least for a more constructive approach to gluing invariants so that the modeller is forced to understand the refinement more intimately.

Therefore, as an alternative approach to invariant discovery, we show how invariants can be chosen by design using the state machine refinement structure as a guide. This simply consists of placing invariants inside the new states introduced in a state machine refinement. The gluing invariant is generally located in the new state which has outgoing final transitions that elaborate an old transition. The incoming transitions represent new preliminary steps leading up to this refined transition. However, it may be necessary to add invariants in other new states where a sequence of new transitions is involved. Placing the invariant inside the state implies that it is true only while in that state and UML-B automatically adds an appropriate antecedent (corresponding to those chosen by heuristics in the discovery method) to this effect. The invariant is chosen by looking at the guards and actions of the incoming transitions to find properties that are true for all incoming transitions. There are two cases; state invariant properties may be true because the incoming transition is only taken when the property is true (and the actions do not change it) or because the transition establishes the property via its actions. Notice that, in the first case, such unchanged properties might not be explicitly mentioned in the guards if they are implied by the source state guard. Therefore, such properties may need to be carried forward from a previous state to the next state. Since the state invariants are derived from the guards and actions of the incoming transitions, they are certainly true and (usually) easily proved. Certainly, any transitions other than the incoming ones will be easily proved since they will clearly negate the antecedent and since the invariants are derived in a simple way from the incoming transitions, there is also a good chance that the automatic provers will find appropriate hypotheses easily. Hence, it is not a problem to be quite liberal in adding these invariants. If we have now defined everything relevant about the source state of the subsequent outgoing refined transition, there should be sufficient information for the proof of the refinement (otherwise, the model must be faulty). The prover easily finds these hypotheses because it contains an instance of the antecedent quantification in its guard (corresponding to the generated source state guard). Of course this only works if there is no interference from other events or state machines that may be in parallel with this one, but that is equally true of invariant discovery. (Such interference may indicate a complex gluing invariant that is not amenable to systematic methods or, more probably, that there is a mistake in the model.) Adding these state invariants is quite easy if you understand the refinement and this is sufficient to allow the prover to prove the PO. Effectively, state invariants like this provide a link between the intermediate state spaces in a sequence of transitions, which is exactly what the prover is lacking. They do this by linking our explicit generated annotation of states to the underlying conceptual state space.

As an example (Fig. 15), we show the gluing invariants constructed for the same event, *withdrawOK*, of *ATM_R3* that featured in the first example of invariant discovery. This is a transition for which we have added some preliminary transitions *requestWD* within a new nested state machine. Examining the guards and actions of *requestWD*, we see that the variables of interest are *atm_cashA* and *atm_wdam* both parameterised for the instance *selfATM*. The parame-

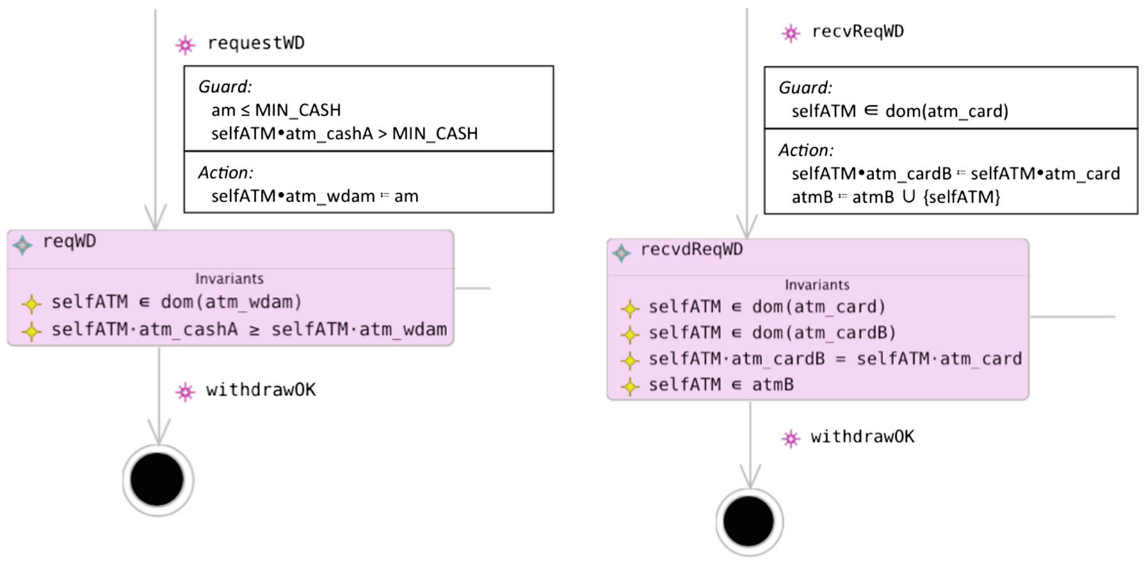


Fig. 15 Gluing invariants by design

ter *am* is local and therefore cannot figure in the invariant. The action sets *selfATM.atm_wdam* to a value, which is less than *selfATM.atm_cashA* and hence the second invariant. We need to know that *atm_wdam* is a partial function (and have some experience of feasibility proof obligations) to realise that *selfATM.dom(atm_wdam)* is important. There are other invariants that we could have derived involving *MIN_CASH* which would have done no harm but turn out to be unnecessary. After adding these state invariants, which are directly equivalent to those added by the discovery method, the proof completed automatically. The second example corresponds to the same event, *withdrawOK*, of *ATM_R4* that featured in another example of invariant discovery. Here, all the guards and actions are reflected as state invariants in a straightforward manner, and again, this was sufficient to automatically discharge all the relevant proof obligations.

In the case of interference from another event, the invariants might be violated. To prevent the violation, we may need to add a guard to the interfering event. For example, consider Fig. 15. If there was another event that was not part of the state machine of this figure that modifies *selfATM.atm_cashA*, then it could violate the invariant $selfATM.atm\_cashA \geq selfATM.atm\_wdam$. An example would be an *empty_cash* event with the action $selfATM.atm\_cashA := 0$. To prevent *empty_cash* from interfering with the invariant above, we could add a guard to *empty_cash*, specifying that it must not happen while there is a transaction in progress in the ATM.

8 Related work

In this section, we outline some of the work related to refinement of UML diagrams. The work on state machines refinement has been introduced by Snook and Walden in [35]. Their work is based on the old version of UML-B [36], which was based on Classical B and has been extended to include translation to an event 'style' of B (which was a precursor to Event-B). They introduced state elaboration and transition elaboration techniques. The semantics of the state machine refinement is given by Event-B. However, we provide a more precise definition of refined state machine and we provide tool support based on UML-B, giving a different model visualisation from the UML diagram symbol used in [35]. We also introduce class refinement techniques which are not dealt with in [35]. In [26], Plaska et al. have suggested a process for refinement involving the application of patterns that are based on the techniques introduced in [35].

The techniques of adding new attributes and associations to a class and adding new classes to a class diagram have been introduced in informal way for refinement of UML class diagram [6], but no formal notation nor formal refinement concept is used. Templates are introduced for attributes and associations to specify the translation of model elements to low-level design and implementation. Bergner et al. [6] has also discussed on possible tool support for the templates. Also, the technique of state elaboration has been introduced in a refinement of UML state diagram [25] again without a formal notion of refinement. Simons [30] has presented four informal refinement rules of state machines. The rules in the refinements are as follows: (1) New states must be substates nested in the abstract states (super-states), (2) new transitions must only connect between the substates, (3) the incoming and outgoing transitions of the super-states must be preserved, and (4) the self-transitions of the super-states must be preserved. Rules (1) and (2) must also be followed in UML-B state machine refinement. These two rules are achieved by applying the state elaboration technique. Rule (3) must also be followed in UML-B for a state machine refinement to be valid. In contrast to Rule (4), in our work, when refining self-transitions, the occurrence of the transitions either can be many times or can be restricted to once. Restriction to once means removing looping behaviour, and this is a valid refinement since we focus on preserving safety, not liveness, in our work. Unlike our work, Simon's work does not involve any formal notion and does not discuss any tool supporting the rules.

There is much more work on combining UML with formal notations, and we now outline some of this. However, unlike our work, none of this work supports refinement in UML to the best of our knowledge. Lano, Clark and Androutsopoulos [15] present the translation of UML-RSDS into Classical B. The work focused on translating class diagrams into B. Each class is translated into a respective B machine. Unlike UML-B, all classes in a class diagram are translated into one Event-B machine. The constraint language used is OCL, whereas we use μB. Idani et al. [10] have investigated the reverse in which they proposed an approach and tool support for the construction of UML diagrams from B specifications. Ledang and Souquiéres have introduced an approach for translating UML state machine diagrams into Classical B in [18]. The translations use the state function representation, whereas UML-B supports both state function and state set representations. Mammar and Laleau [20] have also work on the translation of class and state diagrams into Classical B. Their work is suitable for the development of data-oriented applications in contrast to our work which is suitable for process-oriented applications. Another difference is that the refinements involve the generated abstract B models and there is no concept of refinement in UML, whereas in our work, the refinements involve the class and state machine diagrams. Laleau and Polack [13] have extended the meta-model of UML class diagrams specifically for information system specification. The semantics of the extended meta-model is defined in B invariants. The invariants formalise the associations in the meta-model and rules for integrity constraints. In [14], tools to translate from UML class diagrams into B machines and vice versa have been developed applying the extended meta-model and semantics in [13]. However, [13] and [14] do not deal with refinement. Knapp et al. [12] have investigated the validity of UML state machine refinements by formalising with MTLA [21]. In contrast to our work, their work does not consider state machine hierarchy in refinements. New transitions and states may be introduced in a refined state machine by replacing old states with new states and transitions. We prefer our approach because the relationship between abstract state machines and refinements is clearer. In UML-B, new transitions and states may be added in nested state machines. UML-B is more restrictive, but this makes the refinement pattern simple and clear. Similar to our work, refining self-transitions may be restricted to once as the work does not focus on liveness properties.

Integration work of UML with Z has also been investigated. Moller et al. [23] have integrated the formal method, namely CSP-OZ [8] into UML and Java. A UML profile for CSP-OZ is developed. A UML profile contains an extension mechanism that consists of *stereotype* and *tag* definitions. This profile, which integrates UML and CSP-OZ, is similar to the UML-B profile [31] of previous version of UML-B. In this work, class diagrams, state machines and the UML-RT structure diagrams are translated to CSP-OZ (an integrated formal method) specifications. Amálio et al. [5] also have investigated an integration between UML and Z. They have introduced a framework called UML+Z for building, analysing and refining models based on UML and Z. UML+Z models consist of class, state and object diagrams. The integration work of UML and VDM has been done by Frey [9]. Frey has introduced a methodology where UML and VDM-SL are used together in modelling to take advantage of both notations. Lausdahl et al. [17] have work on a bidirectional translation between UML class diagram and VDM++. The translation of the sequence diagram is done from UML to VDM++. The translations are implemented as a plug-in to the Overture [16] toolsets.

In Sect. 7, we discussed two simple approaches to finding gluing invariants, both of which rely on UML-B state machine refinement. Llano et al. [19] propose a method to discover Event-B gluing invariants using automated theory formation rather than failed proof obligation. Their approach is more complicated than ours, requiring the construction of data tables from simulation traces of the model, but is more general since it does not rely on the style of refinement imposed by UML-B. In [11], Ireland et al. examine failed proof obligations but focus on finding omissions from the model and do not focus on refinement. In contrast, our approach assumes that the model is correct and only attempts to find a gluing invariant.

9 Conclusions

In [28], we have introduced notions of refined class, refined state machine and extended classtype for UML-B. We used these notions to describe the following refinement techniques:

- Add new attributes and associations to a refined class.
- Add new classes in a refinement.
- State elaboration.
- Transition elaboration.
- Add new attributes and associations to an extended classtype.
- Add new classtypes in a refinement.

In this paper, we provide a more extensive account of UML-B refinement techniques. We give a formalisation for UML-B refinement rules and describe the extensions to the UML-B meta-model, which gives precise definition of the notions of refined class, refined state machine and extended classtype. The meta-model is used to extend the UML-B drawing tools. We have applied the UML-B meta-model extensions in the ATM case study. The Rodin tool was used to generate and prove the proof obligations. Based on the ATM case study, we provide two ways of constructing gluing invariants.

One area that currently lacks support in UML-B is parallel state machines. Although UML-B has support for modelling parallel state machines within one refinement level, the transition elaboration mechanism does not allow the parallel state machines to be linked correctly with their parent state machines. In future, we will extend the UML-B meta-model to support refinement of parallel state machines.

Our experience using UML-B for modelling refinements has been that the proof of refinements is comparatively straightforward compared with working directly with Event-B. This may be due to the organisational structures imposed by the patterns generated by UML-B, for example the explicit annotation of states that was discussed in the section on designing gluing invariants. This is an interesting area for future work including extending the approaches to constructing gluing invariants to other refinement patterns of UML-B and comparing ease of proof with equivalent models written directly and freely in Event-B. Further case studies are needed to explore this as well as to further validate the existing techniques and develop new extensions to UML-B refinement.

We are currently working on supporting decomposition concepts in UML-B. Decomposition is needed to ensure scalability of the method. Event-B machines may be decomposed into several submachines in such a way that each submachine can be refined individually while preserving the overall validity of refinement. That is, if the submachines were recomposed, the composition would be a valid refinement of the original machine before it was decomposed. We have previously [29] introduced techniques to refine UML-B state machines in a way that prepares for decomposition and have introduced the concept of a UML-B composed machine to define the composition of a number of UML-B submachines. Our current work is to develop the UML-B composed machine, extending its features to fully support Event-B decomposition techniques in UML-B. We are also extending the UML-B tooling to visualise the UML-B composed machine.

Acknowledgments The authors gratefully acknowledge the financial assistance from the ADVANCE project and the Malaysian government. Butler and snook are funded in part by the EU FP7 ADVANCE Project (ICT 287563), www.advance-ict.eu. Mar Yah is funded by the Ministry Higher Education Malaysia and Universiti Putra Malaysia.We would like to thank the reviewers for their insightful comments and feedback on this paper.

References

1. Abrial, J.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: application to Event-B. J. Fundam. Inf. **77**(1–2), 1–28 (2007)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**, 447–466 (2010)
4. Abrial, J.: Modeling in Event-B—System and Software Engineering. Cambridge University Press, Cambridge (2010)
5. Amálio, N., Polack, F., Stepney, S. : UML + Z: Augmenting UML with Z, In: Software Specification Methods, pp. 81–102, Hermes Science Publishing (2006)
6. Bergner, K., Rausch, A., Sihling, M., Vilbig, A.: Structuring and refinement of class diagrams. In: The 32nd Annual Hawaii International Conference, vol. 6, pp. 6018 (1999)
7. Butler, M., Hallerstede, S.: The rodin formal modelling tool, BCS-FACS Christmas 2007 Meeting. Formal methods in Industry, London (2007)
8. Fischer, C.: CSP-OZ: A Combination of Object-Z and CSP. University of Oldenburg, Germany.Technical report (1997)
9. Frey, P.: Combining UML Use Cases and VDM-SL, Paper for the Seminar in Software Technology at the Institute for Software Technology. IST), Graz University of Technology, Austria (2000)
10. Idani, A. Ledru, L. ,Bert, D.: Derivation of UML class diagrams as static views of formal B developments, In: International Conference on Formal Engineering Methods, pp. 37–51, Springer, Berlin-Heidelberg (2005)
11. Ireland, A., Grov, G., Butler, M.: Reasoned modelling critics: turning failed proofs into modelling guidance. In: International Conference of Abstract State Machines, Alloy, B and Z, LNCS 5977, pp. 189–202. Springer, Berlin-Heidelberg (2010)
12. Knapp, A., Merz, S., Wirsing, M.: Refining Mobile UML State Machines, LNCS3116, pp. 274–288. Springer, Berlin-Heidelberg (2004)
13. Laleau, R., Polack, F.: A rigorous metamodel for UML static conceptual modelling of information systems. In: International Conference on Advanced Information Systems Engineering, LNCS 2068, pp. 402–416. Springer, Berlin-Heidelberg (2001)
14. Laleau, R., Polack, P.: Coming and going from UML to B: a proposal to support traceability in rigorous is development. In: International Conference of B and Z, LNCS2272, pp. 517–534. Springer, Berlin-Heidelberg (2002)
15. Lano, K., Clark, D., Androutsopoulos, K.: UML to B: formal verification of object oriented models. In: International Conference of Integrated Formal Method, pp. 187–206, Springer, Berlin-Heidelberg (2004)
16. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The overture initiative integrating tools for VDM. J. SIGSOFT Softw. Eng. Notes **35**(1), 1–6 (2010)
17. Lausdahl, K.G., Lintrup, H.K.A., Larsen, P.G.: Coupling Overture to MDA and UML. Master Thesis (2008)
18. Ledang, H., Souquiéres, J. : Contributions for modelling UML state-charts in B. In: International Conference of Integrated Formal Methods, LNCS 2335, pp. 109–127, Springer, Berlin-Heidelberg (2002)

19. Llano, M. T., Ireland, A., Pease, A.: Discovery of invariants through automated theory formation, formal aspects of computing, pp. 1–47. Springer, Berlin-Heidelberg (2012)
20. Mammar, A., Laleau, R.: A formal approach based on UML and B for the specification and development of database application. J. Autom. Softw. Eng. **13**(4), 497–528 (2006)
21. Merz, S., Wirsing, M. , Zappe, J.: A spatio-temporal logic for the specification and refinement of mobile systems, LNCS 2621, pp. 87–101. Springer, Berlin-Heidelberg (2003)
22. Metayer, C., Abrial, J.R., Voisin, L.: Event-B language. Technical report deliverable 3.2, EU project IST-511599—RODIN, http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf. Date Last Accessed: 25/1/08 (2005)
23. Moller, M., Olderog, E., Rasch, H. , Wehrheim, H.: Linking CSP-OZ with UML and Java: a case study. In: International Conference of Integrated Formal Methods, LNCS2999, pp. 267–286, Springer, Berlin-Heidelberg (2004)
24. Object Management Group: introduction to OMG's Unified Modelling Language (UML). http://www.omg.org/gettingstarted/what_is_uml.htm. Date Last Accessed:23/8/13
25. Object Management Group: UML 2.1.2 Superstructure Specification. http://www.omg.org/cgi-bin/docs/formal/2007-11-02.pdf. Date Last Accessed:23/8/13
26. Plaska, M., Walden, M., Snook, C. : Documenting the progress of the system development. In: Proceedings of Workshop on Methods, Models and Tools for Fault Tolerance, pp. 251–274, Springer, Berlin-Heidelberg (2007)
27. Rumbaugh, J., Booch, G., Jacobson, I.: The Unified Modelling Language User Guide. Addison Wesley, Reading (1999)
28. Said, M.Y., Butler, M., Snook, C.: Language and tool support for class and state machine refinement in UML-B. In: International Conference of Formal Methods, LNCS 5850, pp. 579–595, Springer, Berlin-Heidelberg (2009)
29. Said, M.Y.: Methodology of Refinement and Decomposition in UML-B, Ph.D. Thesis, University of Southampton, Southampton (2010)
30. Simons, A.J.H.: A theory of regression testing for behaviourally compatible object types. J. Softw. Test. Verif. Reliab. **16**(3), 133–156 (2006)
31. Snook, C., Butler, M., Oliver, I.: The UML-B profile for formal systems modelling in UML, In: UML-B Specification for Proven Embedded Systems Design, pp 69–84, Springer, Berlin-Heidelberg (2004)
32. Snook, C., Butler, M. : UML-B and event-B: an integration of languages and tools. In: The IASTED International Conference on Software Engineering, pp. 336–341 (2008)
33. Snook, C., Fritz, F., Illisaov, A.: An EMF framework for event-B. In: Workshop on Tool Building in Formal Methods—ABZ Conference, Orford, Quebec, Canada (2010)
34. Snook, C., Savicks, V., Butler, M.: Verification of UML models by translation to UML-B. In International Conference of Formal methods for Components and Objects, LNCS6957, pp. 251–266, Springer, Berlin-Heidelberg (2012)
35. Snook, C., Walden, M. : Refinement of statemachines using event B semantics, B2007: formal semantic and development in B. LNCS 4355, pp. 171–185, Springer, Berlin-Heidelberg (2006)
36. Snook, C., Butler, M.: UML-B: formal modelling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (2006)
37. The Eclipse Foundation: Eclipse Modelling Framework. http://www.eclipse.org/emf/. Date Last Accessed: 07/08/2013
38. The Eclipse Foundation: Graphical Modelling Project. http://www.eclipse.org/gmp/. Date Last Accessed: 07/08/2013
39. The Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/. Date Last Accessed: 07/08/2013

Author Biographies

Mar Yah Said is a senior lecturer in the Software Engineering and Information System department of the Faculty of Computer Science and Information Technology at the Universiti Putra Malaysia. She earned her Ph.D. in Computer Science from University of Southampton in 2010, working with Prof. Michael Butler and Dr. Colin Snook on refinement and decomposition methods in UML-B. Her current research extends the decomposition work in UML-B. She has the interest to explore methods for transforming software requirements into a formal specification, which may therefore assist usage of formal modelling. She also interested to discover ways to automate test case derivation from formal specification.

Michael Butler is a Professor of Computer Science at the University of Southampton where he leads the Electronic and Software Systems Research Group. His main research focus is in refinement-based formal methods for complex systems engineering. Butler plays a leading role in the development of several tools for B and Event-B especially the Rodin toolset and he has on-going active collaboration with engineers in several embedded systems industries. He has served as Programme and General Chair of several leading international conferences on formal methods, is a Fellow of the British Computer Society and Chair of IFIP WG 2.3 Programming Methodology.

Colin Snook is a senior research fellow in the Electronic and Software Systems Research Group of the Electronics and Computer Science department at the University of Southampton. He is currently working on the FP7 Advance project developing tools for formal modelling of cyberphysical systems and on the EPSRC Prime project modelling power and reliability management of many-core computers. Dr. Snook is known for the development of the UML-B modelling notation, facilitating the use of formal methods in industry and empirical assessment of formal modelling notations. Dr. Snook gained his Ph.D. at the University of Southampton in 2001. Before that, he worked for Hawker Siddeley Dynamics Engineering (now AT Engine Controls Ltd.) developing real-time embedded control systems in various application domains including avionic engine controls.