Class and State Machine Refinement in UML-B

Mar Yah Said, Michael Butler, and Colin Snook

ECS, University of Southampton, Southampton, SO17 1BJ, UK
(mys05r,mjb,cfs)@ecs.soton.ac.uk

Abstract. UML-B is a 'UML-like' graphical front end for Event-B. It adds support for object oriented modeling concepts while visually retaining the Event-B modeling concepts. In the continuity of the work on UML-B, we strengthen its refinement concepts. Development in Event-B is done through refinements of an abstract model. Since Event-B is reflected in UML-B, the abstraction-refinement concepts must also be catered for in UML-B. UML-B introduced the new concept of refinement, where model complexity is managed by introducing more detailed versions of a machine. We extend this refinement concept by introducing the notion of refined classes and refined state machines. A refined class is one that refines a more abstract class and a refined state machine is one that refines a more abstract state machine. The UML-B drawing tool and Event-B translator are extended to support the refinement concepts. A case study of an auto teller machine (ATM) is presented to demonstrate the notion of refined classes and refined state machines.

Keywords: Visual modeling languages, Formal specification, UML, Event-B, Refinement.

1 Introduction

UML-B [1] is a graphical formal modelling notation that has some resemblance with UML [2, 3] and is based on Event-B [4] which is a new variant of classical B. The UML-B notation is supported by the UML-B tool which is a plug-in extension feature to the Rodin Event-B verification tool [6, 11]. The UML-B tool generates Event-B models corresponding to a UML-B development and the Rodin tool is then used to discharge proof obligations associated with the generated Event-B models. The existing UML-B tool enhanced the old UML-B [12] which is a profile of UML that defines a subset and specialisation of UML. The old UML-B is based on the classical B instead of Event-B.

A development in classical B or Event-B is done through refinement. Refinement [8, 9] is a technique which is used to relate the abstract model of a software system to another model that is more concrete while maintaining the same properties of the abstract model. Refinement is an important technique for managing the complexity of a system being developed. It is important to have an abstract model of a system so that the core functions of a system can be focused on and validated. Further refinements of the abstract model allows the modeler to focus on different aspects of the system at different abstraction levels.

For a development in classical B or Event-B, at the most abstract model, it is required to specify an invariant that defines the static properties of the data being modeled. This invariant must be preserved by all the events of the model. Each refinement will add further invariants relating the abstract model and the refined model. Refinement in classical B or Event-B is done by refining both its state and its events. This is essentially done by extending the list of state variables (possibly suppressing some of them) and by refining each abstract event into corresponding concrete version.

There are two main differences between Event-B and classical B with regards to refinement of events. In Event-B, several events may refine an abstract event whereas in classical B, only one event can refine an abstract event. The other difference is that in Event-B, we may have new events that refine skip whereas in classical B, this is not allowed.

Another difference between classical B and Event-B is that Event-B distinguishes between contexts and machines. A context contains definitions and properties of types and constants. A machine contains state variables, invariants and events that update the variables. A machine may see several contexts.

The previous versions of UML-B did not support refinement. Current work is focused on ways of performing refinement in UML-B. The main contributions of our work are introducing a notion of refined classes and inherited attributes which is described in Section 3 and a notion of refined state machines and refined states which is described in Section 4. The other contribution is introducing a technique of event movement in Section 5.

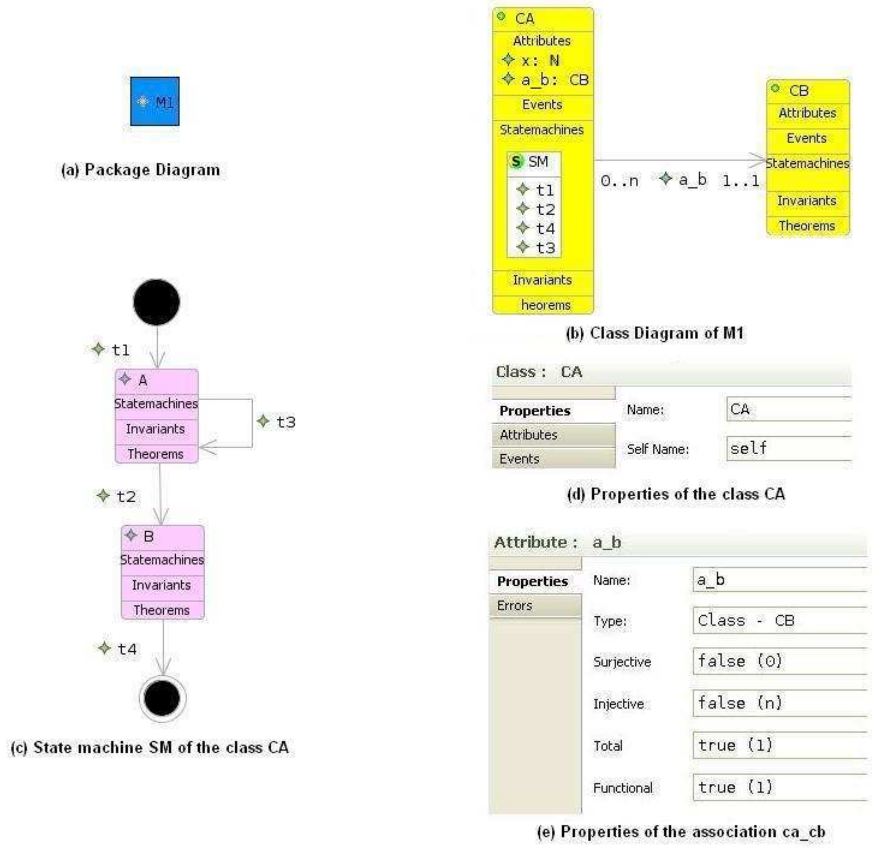
Section 3 describes the technique of performing refinement using the notion of refined classes and inherited attributes which includes adding new classes to a class diagram in a refinement and adding new attributes and associations to refined classes. Section 4 describes the technique of elaborating refined states into sub-states and the transitions elaboration technique. Section 5 describes the technique of moving events from a class in an abstract machine into a new class in a refinement.

Before the technical details of the contributions are describes, we give a background of UML-B and the generated Event-B in Section 2 that outlines the existing features of UML-B that are relevant. Section 6 presents the ATM case study using the refinement techniques describes in Sections 3, 4 and 5. Section 7 concludes the paper.

2 Background of UML-B and Generated Event-B

UML-B provides four kind of diagrams. They are package, context, class and state machine diagrams. A package diagram is a top-level diagram that shows the structure and relationships between components (machines and contexts) in a project. A context is described in a context diagram which is similar to a class diagram but has only constant data and structured types. 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 the class diagram. Events can also be represented by the transitions on a state machine diagram. Further descriptions are focused on the class and state machine diagrams as the rest of the sections mostly concerns these.

A class diagram may contain classes. Each class may have attributes, associations, events and state machines. An attribute defines a data value of an instance of a class. An association is a special case of an attribute that defines a relationships between two classes. An event defines operations of a class and involves modification to some or all the attributes of a class. Class events replace traditional object oriented methods. A state machine defines the behavior of a class in terms of transitions between discrete states. Each UML-B context gives rise to an Event-B context. Each UML-B machine gives rise to both an implicit Event-B context and an Event-B machine. The implicit context is used to define types for the classes and states in the UML-B machine. In the generated Event-B machine classes, class attributes and associations become variables. Events and transitions in classes and state machines become events in the generated Event-B machine.



Class : CA

Name:	CA
Self Name:	self

Attribute : a_b

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

Fig. 1. Package diagram and the UML-B specification of the Abstract Machine M1

Fig. 1 shows an example of a package diagram that contains machine **M1** (a) which has a class diagram (b) containing classes `CA` and `CB`. These classes give rise to the sets `CA_SET` and `CB_SET` in the generated Event-B implicit context. In the generated Event-B machine the classes `CA` and `CB` give rise to variables. The class `CA` consists of the attribute *x* of type $\mathbb{N}$ and also the association *a_b* of type CB. The multiplicity property for the association *a_b* shown in Fig. 1(e) specifies a many-to-one relationship (i.e., total function). A full explanation of association multiplicity may be found in [12]. The attributes *x* and *a_b* also represent variables in the generated Event-B machine. For each class, attribute and association, a type invariant will be generated in the Event-B machine. For example, the class `CA` corresponds to the type invariant which specifies that `CA` is a subset of `CA_SET` ($CA \in \mathbb{P}\,(CA\_SET)$). Attribute *x* corresponds to the type invariant $x \in CA \rightarrow \mathbb{N}$ that specifies *x* is defined for all `CA`. Each class has a self name property with a default value *self*, i.e., the default identifier that represents an instance of a class (which may be changed by the modeler). The self name property of the class `CA` is shown in Fig. 1(d). A class may have events and for each event, its parameters, guards and actions can be defined explicitly as properties. μB (micro B) notation that borrows from the Event-B notation is used for textual guards and actions. μB uses an object-oriented style dot notation to show ownership of entities, i.e., attributes and associations, by classes. Variables used in an expression can represent owned features using the dot notation. For example, *i.x* refers to the value of the variable *x* which belongs to instance *i*. Another example of this will be presented later (Fig. 5).

Attached to the class `CA` is its state machine, `SM`, listing its four transitions *t1*, *t2*, *t3* and *t4*. The state machine `SM` in Fig. 1(c) shows its two states, `A` and `B` and the transitions. The solid black circle is the initial state, whereas, the solid black circle with an outer circle is the final state. The translation to Event-B for a state machine can either be a disjoint sets representation or state function representation. This two styles are introduced in [10] and they are supported in the UML-B tool. UML-B allows modelers to switch between these two representations.

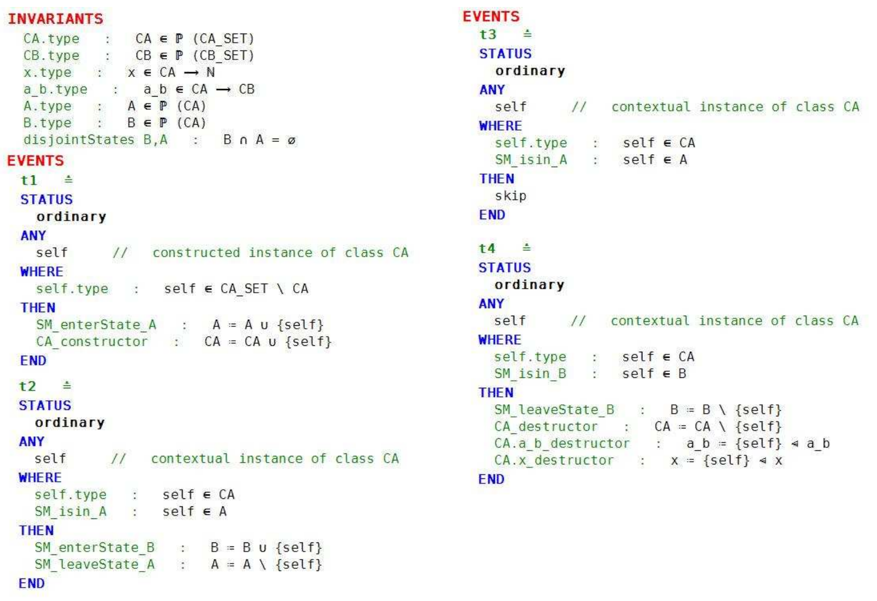
For a disjoint sets representation, a disjoint sets of CA are introduced as variables as follows:

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

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 variable SM (i.e., the state machine belonging to the class `CA`) is introduced representing a function mapping CA to an enumerated set of states, SM_STATES as follows:

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

That is, SM maps each instance of CA to its state. In this paper, the translation to Event-B is described using the disjoint sets representation. The generated Event-B machine for **M1** is shown in Fig. 2. 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 `SM` state machine represent variables of type *CA* (i.e., the state machine owner). An instance of `CA` changes its state when a transition fires. For the states, an additional invariant stating that they are disjoint is added (i.e., $A \cap B = \varnothing$). For each transition there is a guard that specifies an instance source state (labeled as *.._isin_..*) and actions that specify its target state (labeled as *.._enterState_..*) and its departure from the current state (labeled as *.._leaveState_..*). The parameter, *self*, indicates an instance of a class. 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* (labeled *self.type*). A transition to a final state such as *t4* is a destructor which removes an instance from current instances and from the domain of all the class variables. The transition *t3* is a self loop transition which does not changes state. In the generated Event-B the event *t3* has a guard that specifies its source state but with a skip action i.e., not changing state.



```
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  ≙
 STATUS
   ordinary
 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  ≙
 STATUS
   ordinary
 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  ≙
 STATUS
   ordinary
 ANY
   self     //   contextual instance of class CA
 WHERE
   self.type   :   self ∈ CA
   SM_isin_A   :   self ∈ A
 THEN
   skip
 END

 t4  ≙
 STATUS
   ordinary
 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. 2. Generated Event-B specification of M1

Invariants and theorems (assertions requiring proofs) can be attached to classes or states and become part of the Event-B machine. A full explanation and examples of these is in [1].

3 Refinement of Classes in UML-B

In this section, the refinement techniques concerning the notion of refined classes and inherited attributes are described.

The motivation for refined classes and inherited attributes come from performing refinement in Event-B. The notion of refined classes and inherited attributes in UML-B reflect the refinement of variables in Event-B. A refined class is one that refines a more abstract class and an inherited attribute is one that inherits an attribute of the abstract class. A notion of refined classes is needed in UML-B because some elements of an abstract UML-B model need to be retained by the refinement.

In Event-B refinement, a machine that refines another machine (i.e., abstract machine) may keep variables of an abstract machine, may drop some of the old variables and may introduces new variables. In UML-B refinement, a machine (i.e., refined machine) that refines another machine (i.e., abstract machine) may contain refined classes where each refined class refines a class of its abstract machine (i.e., keeps all variables of its abstract machine). In UML-B refinement, a refined machine may drop some of refined classes (i.e., drop some variables). Also in UML-B refinement, a refined machine may introduce new classes (i.e., new variables) in a class diagram.

In UML-B refinement, a refined class that refines a class may inherit attributes of the abstract class (i.e., keeps variables of its abstract machine). A refined class may drop some of the attributes of the abstract class (i.e., drop some variables of its abstract machine) and a refined class may introduce new attributes (i.e., new variables). The following schematic table illustrate a refined class that inherits and drops abstract attributes and introduces new attributes. The table lists out the attributes for class `C` and a refinement of class `C`. Class `C` contain attributes *a1*, *a2* and *a3*. In refinement, the refined class `C` inherits attributes *a1* and *a2*, drops attribute *a2* and has new attributes `a4` and `a5`.

Class C	**Refined Class C**
a1	a1 (*inherited*)
a2	a2 (*inherited*)
a3	a4 (*new*)
	a5 (*new*)

We describe here an example of performing refinement in UML-B using the notion of refined classes and inherited attributes. Fig. 3 shows an example of a package diagram that manages a refinement relationship between machines. The package diagram shows that machine **M2** refines machine **M1**. The class diagram of **M2** is also shown in Fig. 3 where it consists of refined classes `CA` and `CB` which refine the classes `CA` and `CB` of machine **M1** respectively. The refined class `CA` of machine **M2** inherits attribute *x* and association *a_b* of the class `CA` of machine **M1**. The refined class `CB` of machine **M2** has a new association *cb_cc*. Machine **M2** has a new class, `CC` which corresponds to a new set (*CC_SET*) in the generated Event-B implicit context. The refined classes `CA` and `CB` are coloured white to differentiate them from an ordinary class `CC`. In the generated Event-B machine for machine **M2**, the variables `CA`, `CB`, `x` and `a_b` are retained. The machine **M2** has new variables `CC` and `cb_cc` with their type invariants $CC \in \mathbb{P}\ (CC\_SET)$ and $x \in CA \rightarrow \mathbb{N}$ respectively.

In Event-B refinement, a machine that refines another machine (i.e., abstract machine) must provide a refinement of each abstract event. This can be either one event that refines one abstract event or many events that refines one abstract event. New events may be introduced in the refinement. Similarly, in UML-B refinement, at least one concrete event must refines each abstract event and new events may be introduced. These concrete events can either be attached to refined classes or a state machine of a refined class. In UML-B refinement, we can also define additional invariants and theorems by attaching them to refined classes and states that reflect adding invariants and theorems in Event-B refinement.

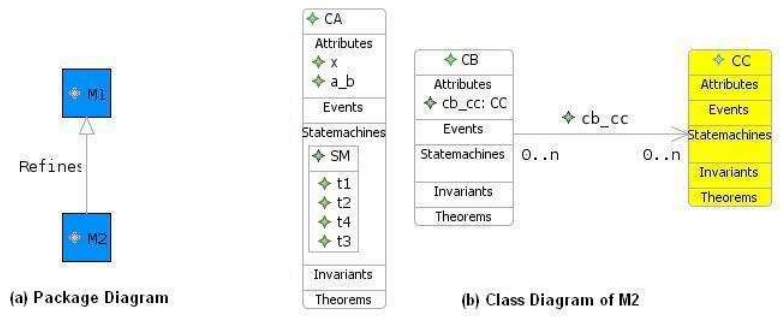


Fig. 3. Package diagram and Class Diagram of Machine M2

4 Refinement of State Machines in UML-B

In this section, the refinement techniques concerning the notion of refined state machines and refined states are described.

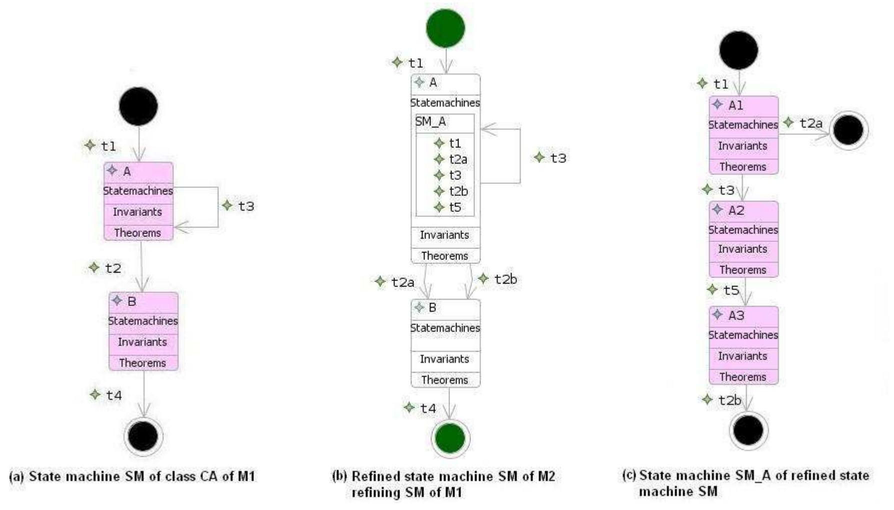


Fig. 4. Refinement of State machine (machine M2 refines machine M1)

The motivation for refined state machines and refined states come from combining the state machine hierarchy in UML-B with refinement in Event-B. The essential concept is that state machines are refined by elaborating an abstract state with nested sub-states. A refined state machine is one that refines a more abstract state machine and a refined state is one that refines a more abstract state.

In UML-B refinement, a refined machine may contains refined state machines and refined states of its abstract machine. We describe first an example of performing refinement in UML-B using the notion of refined state machines and refined states. We will then describe the general rules. Fig. 4 shows an example of refinement of a state machine. The refined class `CA` of **M2** (Fig. 3(b)) has a refined state machine `SM` (Fig. 4(b)) that refines the state machine `SM` of class `CA` of machine **M1** (Fig. 4(a)). The states of refined state machine `SM` are the state `A`, that refines state `A` of machine **M1** and the state `B`, that refines state `B` of machine **M1**. The refined state machine `SM` contains the transitions *t1*, *t2a*, *t2b*, *t3* and *t4* which refine their abstract transitions of machine **M1**. In Fig. 4(b), the abstract transitions *t2* is replaced with transitions *t2a* and *t2b* which refine the abstract transition *t2* of machine **M1**. This refinement of a state machine reflects the refinement in Event-B where many events refine one abstract event. The transitions *t2a* and *t2b* have different source sub-states (i.e., representing different guards in Event-B) which are defined in the nested state machine `SM_A`.

The nested state machine `SM_A` (Fig. 4(c)) elaborates the refined state A (Fig. 4(b)) of machine **M2**. The nested state machine, `SM_A` has three states `A1`, `A2` and `A3`. The transitions *t1*, *t2a*, *t2b* and *t3* in the nested state machine `SM_A` are labelled the same as the incoming and outgoing transitions of the refined state `A`. The same labels indicates that the transition *t1* of the state machine `SM_A` is the same transition as the transition *t1* of the refined state machine `SM` and similarly for *t2a*, *t2b* and *t3*. The transition *t1* of the nested state machine `SM_A` in Fig. 4(c) elaborates the incoming transition *t1* of the super refined state `A`. It means, in the refinement, the target state of the transition *t1* is the state `A1`. The transitions *t2a* and *t2b* of the nested state machine `SM_A` elaborate the outgoing transition *t2a* and *t2b* of the super state `A`. In Fig. 4(b) we do not see a distinction between transitions *t2a* and *t2b*. In Fig. 4(c) we can see a distinction: *t2a* has sub-state `A1` as a source while *t2b* has `A3` as source. The transition *t3* of the nested state machine `SM_A` elaborates the self loop transition of the refined super state `A` specifying its source state as the state `A1` and its target state as `A2`. In the nested state machine `SM_A`, the transition *t5* is a new transition that represents a new event in the generated Event-B machine.

In the generated Event-B machine, type invariants are created for all sub-states, where their types are their super state, for example $A1 \in \mathbb{P}(A)$ is a type invariant for the state `A1`. An additional invariant is generated to specify that all sub states constitute their super state. For example, $\mathtt{A} = \mathtt{A1} \cup \mathtt{A2} \cup \mathtt{A3}$. Other generated invariants are a number of disjointness invariants specifying that all sub states are disjoint.

In the next paragraphs, we give a general definition of state machine refinement based on the given example above. A refined state machine refines a state machine (i.e., abstract state machine) of an abstract class. The structure of a refined state machine is an elaboration of the structure of its abstraction in two possible ways:

- Each transition is replaced by one or more transitions.
- An abstract state may be elaborated by a nested state machine (see below).

In the given example, we used the techniques of state elaboration and transition elaboration. In UML-B refinement, a refined state may be elaborated to sub states contained in a nested state machine forming a state machine hierarchy. State elaboration enables more transitions to be added to a nested state machine. Some of these transitions elaborate the incoming and outgoing transitions of the super state (i.e., the abstract state). Some of these transitions are new transitions (i.e, reflects introducing new events in Event-B).

In UML-B, nested state machines are modeled in separate state machine diagrams from their parent state machine diagram. Therefore, the transition elaboration technique is needed so that transitions in a nested state machine can elaborate the incoming and outgoing transitions of the super state. In a nested state machine, a transition with an initial source state elaborates at most one incoming transition to the super state and a transition with a final target state elaborates at most one outgoing transition from the super state.

An abstract state may have a self loop transition. In UML-B refinement, while the state is elaborated into sub states, the self loop transition may be elaborated as one of the transitions between any two of the sub states. The elaborated transition defines the state changes from a sub state to another sub state when the transition fires.

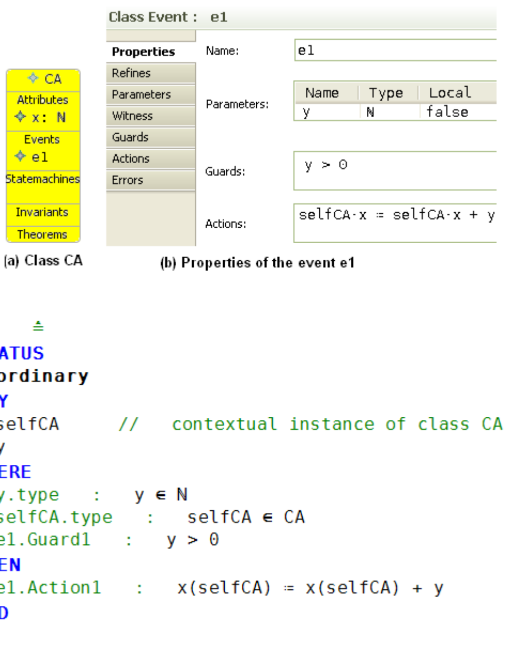
5 Event Movement

This section describes the technique of moving an event from a refined class into a new class introduced in a UML-B refinement. In contrast to the previous described techniques, we are not introducing any new UML-B language feature with this technique. The event movement technique may be used in UML-B refinement when a new class is added and it is natural that the new class is responsible for performing the event transferred from the abstract class to the new class.

We describe the event movement technique with an example shown in Fig. 5 and Fig. 6. The class `CA` in Fig. 5(a) contains an event *e1*. In refinement (Fig. 6(a)), a new class `CC` is introduced and the event *e1* is moved to the class `CC`. In Fig. 5(b), the action statement is using the self name *selfCA* instead of the default self identifier *self*. This is because the default self name, *self* of the `CA` class of the abstract machine is changed to *selfCA* to avoid conflicts with the default self name property of the new class `CC`. The self name property become a parameter in the *e1* event in the corresponding Event-B machine. Translation to Event-B for *e1* event is shown in the Fig. 5(c).

In the refinement machine, the event *e1* becomes a transition between the two states `C1` and `C2` of state machine `CC_SM` (Fig. 6(b)). A parameter, *ca*, of type `CA` is added to the *e1* transition as shown in the property view in the Fig. 6(c). Also, a witness property is defined for the event *e1* which specified that *ca* in the refinement level represents *selfCA* of its abstract level (i.e., $ca = selfCA$). The properties of event *e1* in Fig. 5(c) and Fig. 6(c) show the other parameter y of type $\mathbb{N}$ and also its guard and action which are defined using the μB notation.

The witness property is adapted from Event-B. In Event-B, a witness is used when replacing a parameter of an abstract event with a different parameter in a concrete event in the refinement. The witness is defined by a predicate involving the abstract parameter. Most of the time, this predicate is a simple equality.



(a) Class CA

Class Event : e1

Name:	e1
Parameters:	
Guards:	y > 0
Actions:	selfCA·x ≔ selfCA·x + y

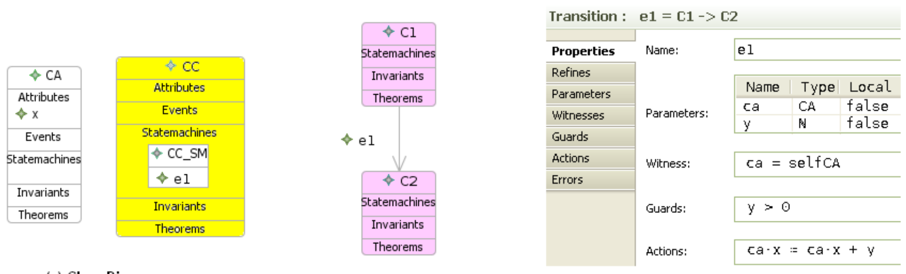
Name	Type	Local
y	ℕ	false

(b) Properties of the event e1

```
e1  ≙
STATUS
  ordinary
ANY
  selfCA     //   contextual instance of class CA
  y
WHERE
  y.type   :   y ∈ N
  selfCA.type   :   selfCA ∈ CA
  e1.Guard1   :   y > 0
THEN
  e1.Action1   :   x(selfCA) ≔ x(selfCA) + y
END
```

(c) Event-B translation of the event e1

Fig. 5. Example of the UML-B specification with an event in a class CA of an abstract machine



(a) Class Diagram

(b) State machine CC_SM

Transition : e1 = C1 -> C2

Name:	e1
Parameters:	
Witness:	ca = selfCA
Guards:	y > 0
Actions:	ca·x ≔ ca·x + y

Name	Type	Local
ca	CA	false
y	ℕ	false

(c) Properties of the event e1

Fig. 6. Example of the UML-B specification when moving an event of a class in an abstract machine into a new class in its refinement machine

This event movement technique is useful when introducing a new class in UML-B refinement. Section 5 will demonstrate its usefulness.

6 ATM Case Study

A case study based on an auto teller machine (ATM) was undertaken to validate the extension of UML-B with regards to the notion of refined classes and refined state machines. An ATM is a machine that allows bank customers to do some of the banking transactions 24 hours per day. It allows bank customers to perform a range of functions, including withdraw cash, check account balance and print mini-statements. In order to perform these functions through an auto teller machine, bank customers need to use their ATM cards which are provided to them by the bank. The case study focused only on the requirements for the cash withdrawal function. There are three machine levels for the ATM UML-B development. These machines are linked by a refinement relationship. The summary for each machine level is as follows:

Abstract machine (**ATM_A**): Models bank accounts and operations on accounts.

First Refinement (**ATM_R1**): Introduces the ATMs, cards and PIN numbers.

Second Refinement (**ATM_R2**: Introduces an explicit validation transition for cards and splits withdrawal into a bank transition and an ATM transition.

The package diagram in Fig. 7 shows a refinement relationship between the machines,

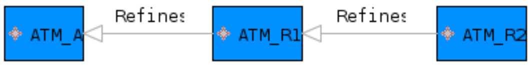
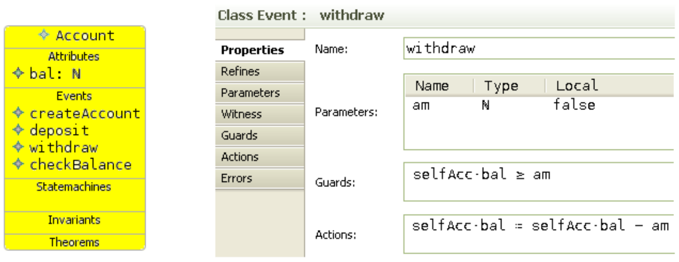


Fig. 7. ATM Package Diagram

Fig. 8 shows a UML-B specification of the ATM abstract machine. The abstract machine consists of a class `Account` (8(a)) 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 specification of the withdraw event is shown in 8(b) including parameters, guards and actions. The *withdraw* event has one added parameter, *am* of type natural number. The parameter is shown in the property view in Fig. 8 including the guard and action. *selfAcc* is the self name property defined for the class `Account`. The *withdraw* event can only occur if the amount, *am*, is less than or equal to the balance in the account. The *withdraw* event will result in decreasing the balance of the account by *am* amount.

The first refinement of the ATM model introduces three new classes which are `ATM`, `Card` and `Pin` which represent the sets of ATMs, ATM cards and PIN numbers respectively. The UML-B specification is shown in Fig. 9. The class diagram (Fig. 9(a)) of machine `ATM_R1` contains the new classes and a refined class `Account` that refines



Class Event : withdraw

Properties | Refines | Parameters | Witness | Guards | Actions | Errors

Name: withdraw

Parameters:

Name	Type	Local
am	ℕ	false

Guards: selfAcc·bal ≥ am

Actions: selfAcc·bal ≔ selfAcc·bal − am

(a) Class Diagram

(b) Properties of the event withdraw

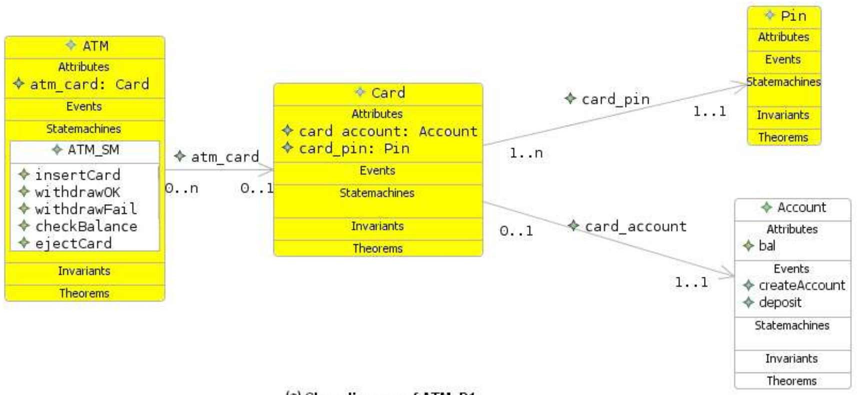
Fig. 8. UML-B specification of ATM abstract machine

the `Account` class of **ATM_A**. The class `ATM` has an association *atm_card* with the class `Card`. The class `Card` has an association *card_pin* with the class `Pin` and an association *card_account* with the refined class `Account`. The refined class inherits the *bal* attribute and 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 an event of the ATM class.

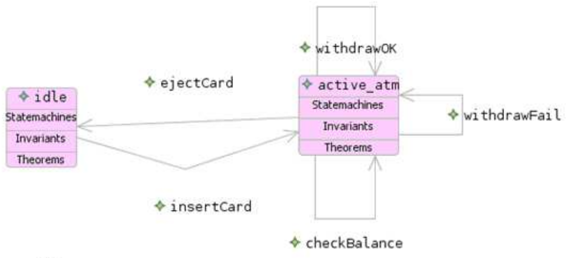
The state machine `ATM_SM` in Fig. 9(b) partitions the behavior of the ATM into either an *idle* state, (i.e., not being used/not active) or *active_atm* state (i.e., is being used). An ATM changes its state when it is triggered by a transition. The *insertCard* transition can occur when an ATM is in the *idle* state and the card inserted is a valid ATM card. When it occurs it changes an ATM state from *idle* to *active_atm*. The *ejectCard* transition changes an ATM state from *active_atm* to *idle*. While an ATM is in *active_atm* state, it means, 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 the withdrawal amount exceeds the account balance.

Fig. 9(c) shows the properties of the *withdrawOK* transition with the parameters, witness, guards and action. The witness specifying that the parameter *ac* represents the *selfAcc* parameter of the abstract *withdraw* event. In this refinement, the guards are strengthened so that the *withdrawOK* transition can only occur when an ATM card is inserted ($selfATM.atm\_card = c$), an instance of ATM is being used ($selfATM \in dom(atm\_card)$) and the ATM card corresponds to a valid account($c.card\_account = ac$). Fig. 9(d) also shows the refines property of the *withdrawOK* transition.

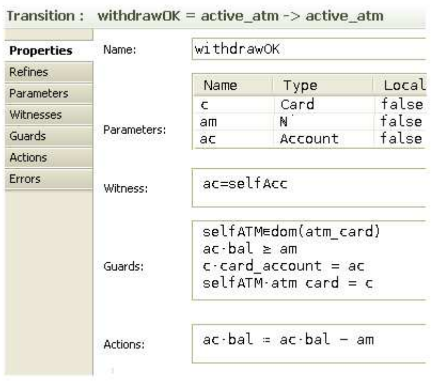
The second refinement models an explicit validation transition for cards and splits withdrawal and balance check into a bank transition and an ATM transition. This is achieved by elaborating the *active_atm* state into sub-states. The class diagram of ma-



(a) Class diagram of ATM_R1



(b) State machine ATM_SM of class ATM of machine ATM_R1



Transition : withdrawOK = active_atm -> active_atm

Name: withdrawOK

Parameters:

Name	Type	Local
c	Card	false
am	ℕ	false
ac	Account	false

Witness:

```
ac=selfAcc
```

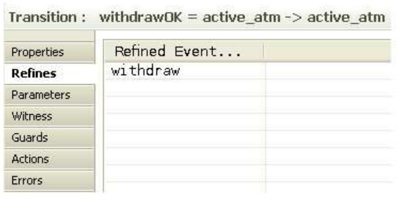
Guards:

```
selfATM∈dom(atm_card)
ac·bal ≥ am
c·card_account = ac
selfATM·atm card = c
```

Actions:

```
ac·bal ≔ ac·bal − am
```

(c) Properties of withdrawOK event



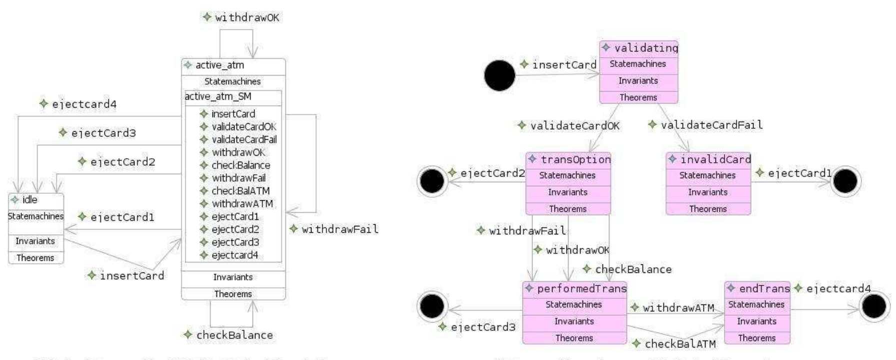
Transition : withdrawOK = active_atm -> active_atm

Refined Event...
withdraw

(d) Refines property of withdrawOK event

Fig. 9. UML-B specification of ATM First Refinement

chine **ATM_R2** (not shown in any figure) contains four refined classes that refine the classes *Account*, *ATM*, *Pin* and *Card* of **ATM_R1** machine. An attribute *atm_cash*, which represents the amount of cash stored in an ATM is added to the refined class `ATM` of machine **ATM_R2**. The refined class `ATM` of **ATM_R2** contains the refined state machine `ATM_SM` which contains the two refined states that refine the states *idle* and *active_atm* of the state machine `ATM_SM` of **ATM_R1** (Fig. 10(a)).



(a) Refined State machine ATM_SM of Refined Class ATM of machine ATM_R2

(b) State machine active_atm_SM of Refined State active_atm of Refined Class ATM of machine ATM_R2

Fig. 10. UML-B specification of ATM Second Refinement

A new state machine named `active_atm_SM` is added to the refined state *active_atm* of **ATM_R2** and it contains five sub-states, namely, *validating*, *invalidCard*, *transOption*, *performTrans* and *endTrans* (Fig. 10(b)). The state machine has a transition *insertCard* which elaborates the incoming transition to the super refined state `active_atm` of **ATM_R2**. The outgoing transitions *ejectCard1*, *ejectCard2*, *ejectCard3* and *ejectCard4* from the states *invalidCard*, *transOption*, *performedBankTrans* and *endTrans* respectively elaborate the outgoing transitions of the super refined state *active_atm* of **ATM_R2**. The transitions *withdrawOK*, *withdrawFail* and *checkBalance* elaborate the self loop transitions of the super refined state *active_atm*. The transitions *validateCardOK*, *validateCardFail*, *withdrawATM* and *checkBalATM* are new transitions.

The state machine refinement in the second refinement introduces an additional level in the state machine nesting hierarchy. This supports modular reasoning, since refinement invariants are only required for the states that are being elaborated, so it localizes proof effort.

All the proof obligations for the three machines **ATM_A**, **ATM_R1** and **ATM_R2** were generated and proved using the Rodin tool provers [6]. The total number of proof obligations (POs) is 200 and most of them are discharged automatically except for three POs in **ATM_R2**. These three POs were proved interactively.

7 Conclusions

UML-B provides a graphical front-end for Event-B. The work described here is a continuation of work for the UML-B, where we provide a way of performing refinement in UML-B. We have described a notion of refined class and refined state machine. We also described the following five refinement techniques:

- Add new classes to a refined class diagram and add attributes and associations to a refined class
- State elaboration
- Transition elaboration
- Move event from a class in a class diagram to a new class in a refined class diagram

Some of the techniques used here (state elaboration, transition elaboration) were previously introduced by Snook and Walden [13]. However, we provide tool support based on UML-B giving a different modeling visualization from the UML diagram symbols used in [13]. We also combine state machine refinement with class refinement techniques, which are not dealt with by Snook and Walden. In [14], Plaska, Walden and Snook suggest a process for refinement involving the application of patterns that are based on the techniques introduced in [13].

The techniques of adding new attributes and associations to a class and adding new classes to a class diagram have been introduced in a refinement of UML class diagram [16]. Also, the technique of state elaboration has been introduced in a refinement of UML state diagram [15].

We have presented the use of the above listed techniques in the ATM case study which was modeled using the UML-B tool. The Rodin tool provers were used to generate and prove the proof obligations. The approach of elaborating states with sub states in refinement, as illustrated by the ATM case study, supports an incremental refinement approach. The hierarchical structure of nested state machines also supports modular reasoning by localising the invariants required for refinement proofs.

The work described here is still in progress. The development of the ATM case study in UML-B will be proceed to the next refinement level. We believe that the result of our research will be a methodology of refinement in UML-B which will assists modeling in UML-B.

Acknowledgements. This material is based upon work supported by the Deploy Project, which is an FP7 Integrated Project supported by European Commission (Grant N 214158). The first author is funded by the Malaysian Government, IPTA Academic Training Scheme and University Putra Malaysia (UPM).

References

1. Snook, C. and Butler, M. : UML-B and Event-B: an integration of languages and tools. In: The IASTED International Conference on Software Engineering - SE2008, 12-14th February 2008, innsbruck, Austria.(2008)
2. Object Management Group (2007). Introduction to OMG's Unified Modelling Language (UML). Online. Date Last Accessed:25/1/08.

3. Rumbaugh, J., Booch, G. and Jacobson, I. : The Unified Modelling Language User Guide. Addison Wesley. (1999)
4. 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)
5. Rigorous Open Development Environment for Complex Systems (RODIN) - IST 511599, http://rodin.cs.ncl.ac.uk/. Date Last Accessed: 25/1/08.
6. Abrial, J. R., Butler, M., Hallerstede, S. and Voisin, L : An Open Extensible Tool Environment for Event-B, Proceedings of ICFEM 2006, LNCS volume 4260/2006, pp. 588-605. (2006)
7. Evans, N. and Butler, M : A Proposal For Records in Event-B, In Journal T. Nipkow an J. Misra, editors Formal Methods 2006, vol 4085, pp. 221-235, Springer-Verlag, McMaster, Canada. (2006)
8. Abrial, J. : The B-Book: Assigning Programs to Meanings, Cambridge University Press (1996)
9. Abrial, R. and Hallerstede, S. : Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B, Journal Fundamentae Informatica. volume 77, pp. 1-28., IOS Press. (2007).
10. Butler, M. and Yadav, D. : An Incremental Development of the Mondex System in Event-B, volume 20, number 1, Springer, journal Formal Aspects of Computing, pp. 61-77. (2008)
11. Butler, M. and Hallerstede, S.: The Rodin Formal Modelling Tool, BCS-FACS Christmas 2007 Meeting, Formal Methods In Industry, London.(2007)
12. Snook, C. and Butler, M. : UML-B: Formal modelling and design aided by UML, ACM Transactions on Software Engineering and Methodology, volume 15, pp. 92-122, ACM Press. (2006).
13. Snook, C. and Walden. M. : Refinement of Statemachines Using Event B Semantics, B2007: Formal Semantic and Development in B, LNCS volume 4355/2006, pp. 171-185, Springer. (2006).
14. M. Plaska, M Walden and C. Snook: Documenting the Progress of the System Development. In Proc. of Workshop on Methods, Models and Tools for Fault Tolerance, Oxford, UK. (2007).
15. OMG: UML 2.1.2 Superstructure Specification. (2007) http://www.omg.org/cgi-bin/docs/formal/2007-11-02.pdf
16. Bergner, K. Rausch, A. Sihling, M. Vilbig, A. : Structuring and refinement of class diagrams: Proceedings of the 32nd Annual Hawaii International Conference, Volume: Track6. (1999)