Language and Tool Support for 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 that provides support for object-oriented modelling concepts. In particular, UML-B supports class diagrams and state machines, concepts that are not explicitly supported in plain Event-B. In Event-B, refinement is used to relate system models at different abstraction levels. The same abstraction-refinement concepts can also be applied in UML-B. This paper introduces the notions of refined classes and refined state machines to enable refinement of classes and state machines in UML-B. Together with these notions, a technique for moving an event between classes to facilitate abstraction is also introduced. Our work makes explicit the structures of class and state machine refinement in UML-B. The UML-B drawing tool and Event-B translator are extended to support the new refinement concepts. A case study of an auto teller machine (ATM) is presented to demonstrate application and effectiveness of refined classes and refined state machines.

Keywords: Visual modelling 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 [8]. UML-B supports class diagrams and state machines, concepts that are not explicitly supported in plain Event-B. The UML-B notation is supported by the UML-B tool which is a plug-in feature for 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. As detailed in [12], our motivations for developing UML-B are twofold. Firstly, in our experience industrial users find the UML-like language and tool appealing. Secondly, UML-B provides additional complementary structuring of Event-B models in the form of classes and state machines.

A development in classical B or Event-B is performed 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 properties of the abstract model. Refinement is an important technique for managing the complexity of a system being developed.

At the most abstract level of a refinement-based development, it is usual to specify invariants that define the properties of the system being modelled. These invariants

This work has been presented at the IM_FMT 2009 workshop of the IFM2009 conference, Dusseldorf, Germany on 16 February 2009.

must be preserved by all the events of the model. Each refinement step will add further invariants relating the abstract model and the refined model (gluing invariants). In Event-B, both state and events may be refined. This is achieved by extending the list of state variables (possibly suppressing some of them) and by replacing each abstract event by corresponding concrete events.

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.

UML-B incorporates the Event-B machine construct and a single UML-B machine may contain multiple classes and multiple state machines. Previously, UML-B supported machine refinement (a refinement relationship between UML-B machines) but had no support for refinement of classes or state machines (which are nested within UML-B machines). The work reported here enriches UML-B to support class and state machine refinement. The main contributions of our work are introducing notions of refined classes and inherited attributes which are described in Section 3 and notions of refined state machines and refined states which are described in Section 4. The other contribution is introducing a technique of event movement in Section 5. A further contribution is that we have implemented the UML-B extensions in the UML-B tool. In this work we focus on safety-preserving refinement and do not deal with liveness.

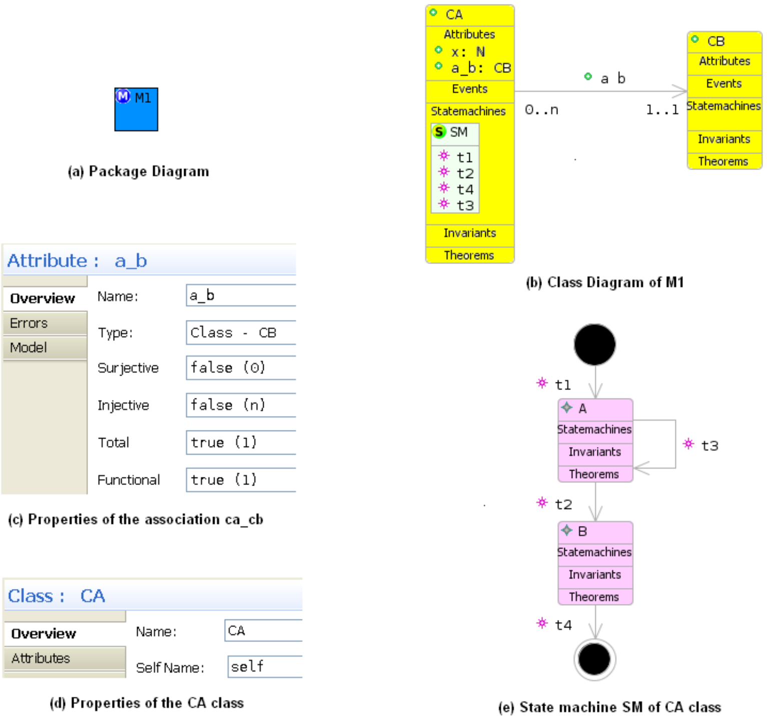
Section 3 describes class refinement using the notion of refined classes and inherited attributes which includes techniques for adding new classes and adding new attributes and associations to refined classes in a refinement. Section 4 describes state machine refinement and a technique for elaborating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

Before the technical details of the contributions are describes, we give some background on UML-B and the generated Event-B in Section 2 that outlines the existing relevant features of UML-B. 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 a class diagram. Events can also be represented by the transitions in a state machine diagram. Further descriptions focus on the class and state machine diagrams as the rest of the sections mostly concerns these. 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.

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. Events and state machines may modify some or all the attributes of any class. Each UML-B context gives rise to an Event-B context (i.e., the UML-B tool generates a corresponding 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.



(a) Package Diagram

(b) Class Diagram of M1

Attribute : a_b

Overview
Errors
Model

Name: a_b

Type: Class - CB

Surjective false (0)

Injective false (n)

Total true (1)

Functional true (1)

(c) Properties of the association ca_cb

Class : CA

Overview
Attributes

Name: CA

Self Name: self

(d) Properties of the CA class

(e) State machine SM of CA class

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

Fig. 1 contains screenshots from the UML-B tool showing 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 ℕ and also the association *a_b* of type CB. The multiplicity property for the association *a_b* shown in Fig. 1(c) 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* give rise to 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 modeller). 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 [12] 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(e) shows its two states, A and B and the transitions. The solid circle is the initial state, whereas, the solid 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. These two styles are introduced in [10] and they are supported in the UML-B tool. UML-B allows modellers to switch between these two representations.

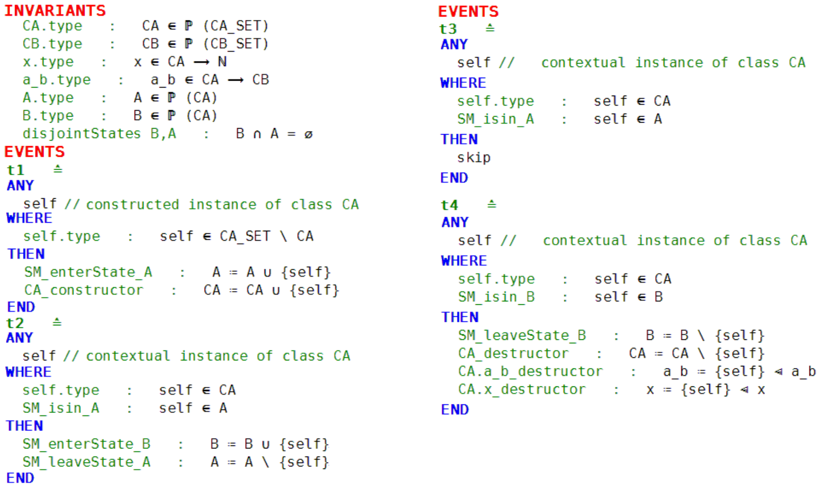
For a disjoint sets representation, a disjoint sets of CA are introduced as variables as follows:

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

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 the Rodin screenshot of 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 generated (i.e., $A \cap B = \varnothing$). For each transition there is a guard that specifies an instance source state (labelled as *.._isin_..*) and actions that specify its target state (labeled as *.._enterState_..*) and its departure from the current state (labelled 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* (labelled *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 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].



```
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. 2. Generated Event-B specification of M1

3 Refinement of Classes in UML-B

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

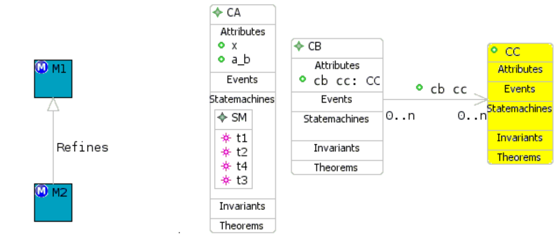
The motivation for refined classes and inherited attributes come from performing refinement in Event-B. The notion of refined classes and inherited attributes in UML-B reflect the refinement of variables in Event-B. A refined class is one that refines a more abstract class and an inherited attribute 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 a more abstract machine may keep variables of an abstract machine, may drop some of the variables and may introduce new variables. In UML-B refinement, a machine that refines a more abstract machine may contain refined classes where each refined class refines a class of its abstract machine (i.e., keeps variables of its abstract machine). In UML-B refinement, a machine may drop some of refined classes (i.e., drop some variables). Also in UML-B refinement, a machine may introduce new classes (i.e., new variables) in a class diagram.

In UML-B refinement, a refined class may inherit attributes of its abstract class (i.e., keeps variables of its abstract machine). A refined class may drop some of the attributes of its 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 illustrates a refined class that inherits and drops abstract attributes and introduces new attributes. The table lists the attributes for class C and a refined class C. Class C contains attributes *a1*, *a2* and *a3*. In refinement, the refined class C inherits attributes *a1* and *a2*, drops attribute *a3* and has new attributes *a4* and *a5*. In the generated Event-B machine, both a class and a refined class give rise to variables. A type invariant is generated for an abstract class i.e., Class C but not for a refined class because its type is already defined in the abstract Event-B machine. Similarly both the inherited attributes and new attributes give rise to variables and a type invariant is generated for each new attribute but not for the inherited attributes.

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

We describe here a simple example of performing refinement in UML-B using the notion of refined classes and inherited attributes. Fig. 3(a) shows an example of a package diagram that manages a refinement relationship between machines. The package diagram shows that machine **M2** refines machine **M1** (of Fig. 1). The class diagram of **M2** is shown in Fig. 3(b) where it consists of refined classes CA and CB refining the classes CA and CB of **M1** respectively. The refined class CA inherits attribute x and association a_b. The refined class CB has a new association cb_cc. Machine **M2** has a new class, CC which gives rise to a new set (CC_SET) in the generated Event-B implicit context. 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 $cb\_cc \in CB \rightarrow CC$ respectively.



(a) Package Diagram

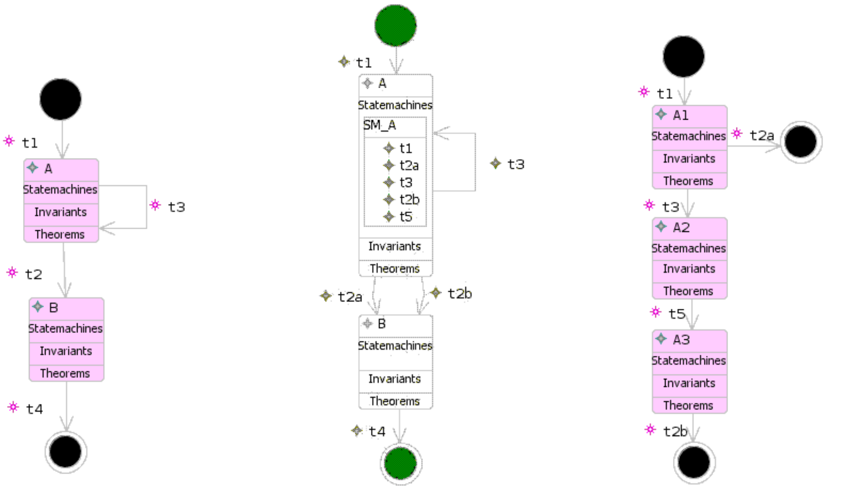
(b) Class Diagram of M2

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

In Event-B refinement, a machine must provide a refinement of each abstract event. This can be either one or many event(s) refining one abstract event. New events may be introduced in a refinement. Similarly, in UML-B refinement, at least one concrete event must refine each abstract event and new events may be introduced. These concrete events can either be attached to a refined (or a new) class or a state machine of a refined (or a new) 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.

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. 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.



(a) State machine SM of class CA of M1

(b) Refined state machine SM of M2 refining SM of M1

(c) State machine SM_A of refined state machine SM

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

In UML-B refinement, a machine may contain refined state machines and refined states. 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 a state machine refinement. The refined class `CA` of **M2** (Fig. 3(b)) has a refined state machine `SM` (Fig. 4(b)) refining the state machine `SM` of **M1** (Fig. 4(a)). The states of refined state machine `SM` are refined state `A` and refined state `B` refining state `A` and state `B` of **M1**. The refined state machine `SM` contains the transitions *t1*, *t2a*, *t2b*, *t3* and *t4* which refine the corresponding abstract transitions of machine **M1**. In Fig. 4(b), the abstract transition *t2* is replaced with transitions *t2a* and *t2b* which refine the abstract transition *t2* of machine **M1**. This refinement of a state machine reflects refinement in Event-B where many events can 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 **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 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 refined super-state A. This means, in the refinement, the target state of the transition *t1* is the sub-state A1. The transitions *t2a* and *t2b* of the nested state machine SM_A elaborate the outgoing transition *t2a* and *t2b* of the refined 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 a 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 representing 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, A = A1 ∪ A2 ∪ 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 example given above. A refined state machine refines a more abstract state machine. 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 refined super-state. Some of these transitions are new transitions (i.e, reflects introducing new events in Event-B refinement).

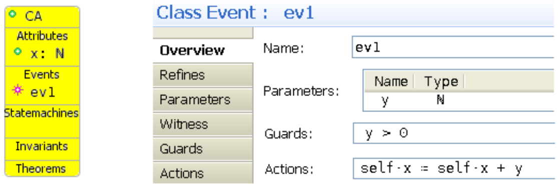
In UML-B, nested state machines are modelled in separate state machine diagrams from their parent state machine diagrams. Therefore, the transition elaboration technique is needed so that transitions in a nested state machine can elaborate the incoming and outgoing transitions of the super-state. In 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. Our experience is that having separate diagrams for nested state machines scales better than embedding them directly in a single diagram. A single diagram can result in scalability problems when there are many levels of state machine hierarchy and a nested state machine has many states. On the other hand, it can be useful to see at least one level of nesting in a single diagram. We will investigate this in future.

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. When refining a self loop transition, the occurrence of the transition can either 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 current work.

5 Event Movement

This section describes the technique of moving a class event in UML-B refinement. There are two methods of moving a class event in a refinement, these are (1) move to a refined class as a transition of a state machine and (2) move to a new class in a refinement either as a class event or a transition in a state machine. Method (1) does not need any new UML-B language feature. However, method (2) creates a motivation for the need to be able to change the default *self* name in UML-B.

We describe both methods by giving first an example of an abstract machine in which a refinement is based upon. Figure 5(a) shows a class `CA` with attribute *x* and event *ev1*. Figure 5(b) shows the properties of the event *ev1* showing its parameter *y*, a guard and an action. The action is defined using μB notation and uses a default identifier *self*, i.e., the self name property which represents an instance of a class `CA`. The self name property becomes a parameter of the *ev1* event in the corresponding Event-B machine.



Name: ev1

Parameters:

Name	Type
y	ℕ

Guards: y > 0

Actions: self·x ≔ self·x + y

(a) Class CA

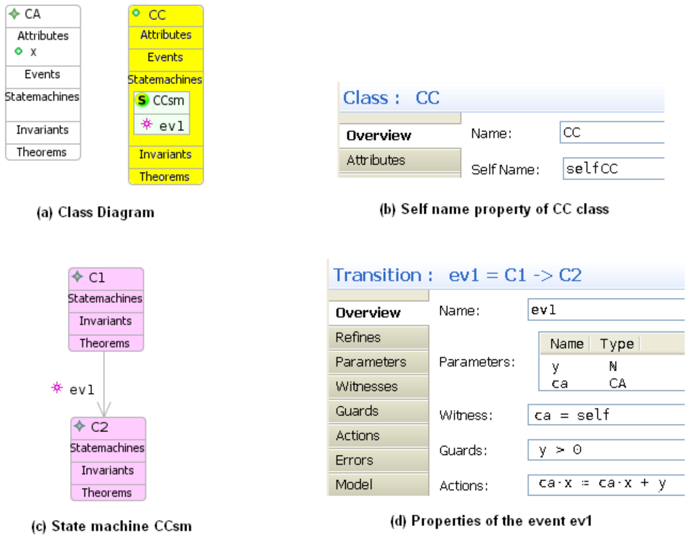
(a) Properties of event ev1

Fig. 5. Example of the UML-B specification of an abstract machine

For method (1), in a refinement, the class event of the abstract machine may be moved to a state machine of the refined class `CA` as a transition *ev1* between the states `A1` and `A2`. In the generated Event-B machine for the event *ev1*, an additional guard specifying the current state `A1` for the event to take place ($self \in A1$) and also additional actions specifying an instance move to the state `A2` ($A2 := A2 \cup \{self\}$) and an action specifying an instance leaves the current state `A1` ($A1 := A1 \setminus \{self\}$) are generated.

For method (2), in a refinement, a class event may be moved to a new class as a class event or as a transition in a state machine. We describe here the event movement technique when a class event is move to a new class as a transition in a state machine. Assume that the UML-B specification in Fig. 6 is a refinement of the abstract machine in Fig. 5. In the refinement, a new class `CC` is introduced and event *ev1* is moved to the class `CC` (Fig. 6(a)). The event *ev1* become a transition between the states `C1` and `C2` of state machine `CCsm` (Fig. 6(c)).

The self name property of the class `CC` is changed to *selfCC* from the default *self* (Figure 6(b)). This change is necessary to avoid conflicts with the default *self* name of the refined class `CA`. In a class event or a transition refining an abstract class event, a parameter whose type is the abstract class is introduced to replace a *self* parameter of the abstract class. For example, a parameter *ca*, of type `CA` is added to the *ev1* transition as shown in the property view in the Figure 6(d). A witness property is defined for the transition *ev1* which specifies that *ca* in the refinement level represents *self* of its abstract level (i.e., *ca* = *self*).



(a) Class Diagram

Class : CC

Name: CC

Self Name: selfCC

(b) Self name property of CC class

(c) State machine CCsm

Transition : ev1 = C1 -> C2

Name: ev1

Parameters:

Name	Type
y	ℕ
ca	CA

Witness: ca = self

Guards: y > 0

Actions: ca·x := ca·x + y

(d) Properties of the event ev1

Fig. 6. Example of the UML-B refinement for the second method

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.

Section 6 of the ATM case study will demonstrate the usefulness of moving events from one class to another in refinement. In the first refinement of the ATM case study, the *withdraw* event of the `Account` class is moved to the `ATM` class as a transition in a state machine of the class `ATM`.

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 ATM, bank customers need to use their ATM cards which are provided to them by the bank. The case study focused on the requirements for the cash withdrawal and check balance functions. There are seven levels for the ATM UML-B development. These machines are linked by a refinement relationship. We described in details the first three levels and described briefly the other four levels. The summary for the first three machine level is as follows:

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

First Refinement (**ATM_R1**): Introduces the ATMs and ATM cards.

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.

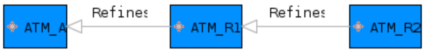
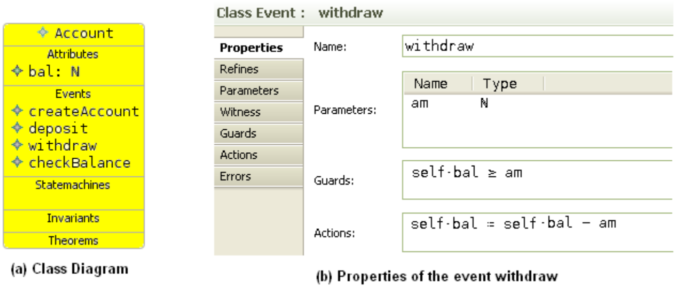


Fig. 7. ATM Package Diagram

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 *withdraw* event has one added parameter, *am* of type natural number. The parameter is shown in the property view in Fig. 8(b) including the guard and action. *self* 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.



Class Event : withdraw

Name: withdraw

Parameters:

Name	Type
am	ℕ

Guards: self·bal ≥ am

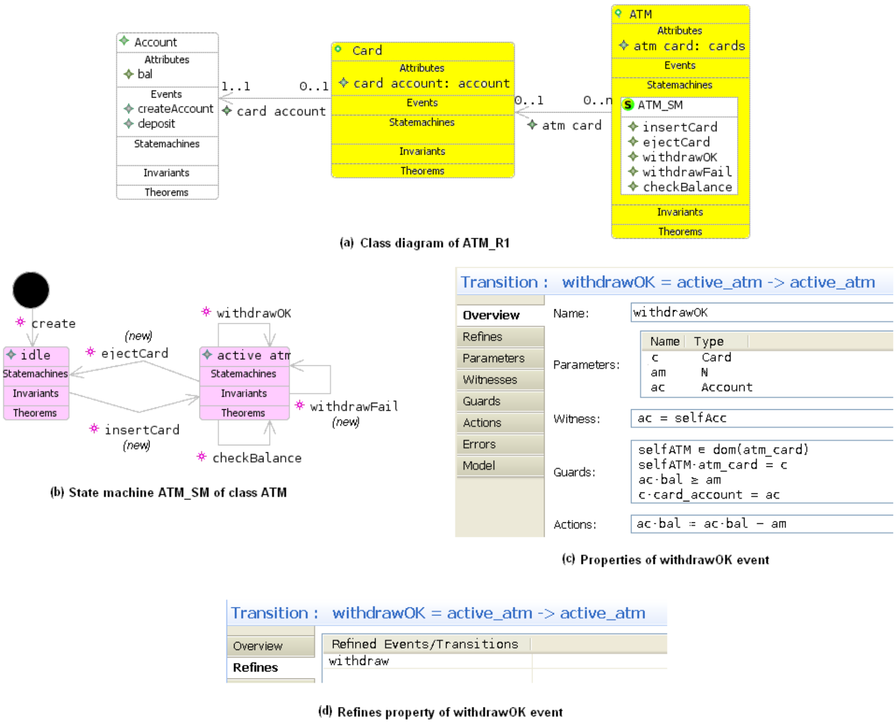
Actions: self·bal ≔ self·bal − am

(a) Class Diagram

(b) Properties of the event withdraw

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

The first refinement of the ATM model introduces two new classes which are `ATM` and `Card` which represent the sets of ATMs and ATM cards respectively. The UML-B specification is shown in Fig. 9. The class diagram (Fig. 9(a)) of `ATM_R1` contains the new classes and a refined class `Account` refining 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_account* with the refined class `Account`. The refined class inherits the *bal* attribute and refines the two events, namely, *createAccount* and *deposit* of `ATM_A`. The other two events 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 (Fig. 8), we specify the effect of a withdrawal on the account balance. In the refinement (Fig. 9), 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.



(a) Class diagram of ATM_R1

(b) State machine ATM_SM of class ATM

Transition : withdrawOK = active_atm -> active_atm

Name: withdrawOK

Parameters:

Name	Type
c	Card
am	ℕ
ac	Account

Witness: ac = selfAcc

Guards:

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

Actions: ac·bal ≔ ac·bal − am

(c) Properties of withdrawOK event

Transition : withdrawOK = active_atm -> active_atm

Refined Events/Transitions
withdraw

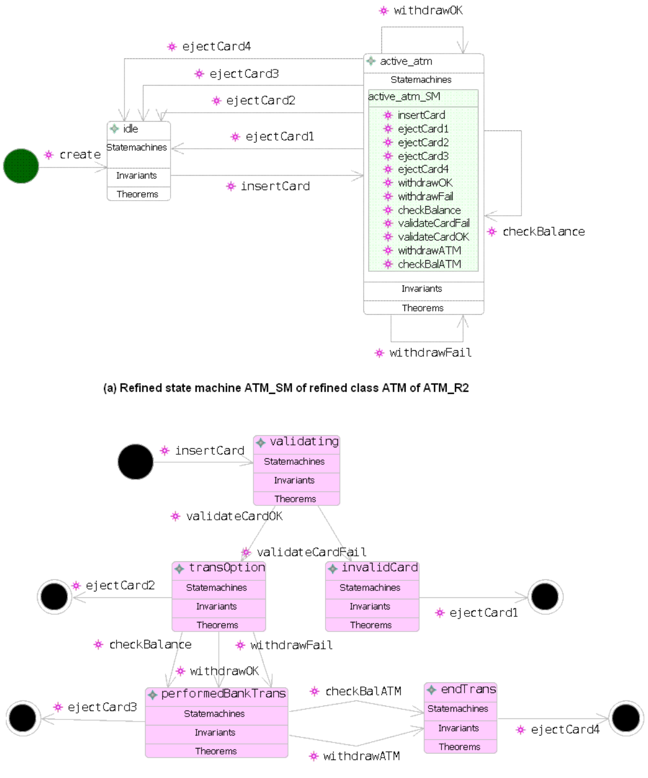
(d) Refines property of withdrawOK event

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

The state machine ATM_SM in Fig. 9(b) partitions 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). An ATM changes its state when it is triggered by a transition. The *create* transition creates a new instance of an ATM and sets its state as *idle*. 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, 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 transitions *withdrawOK* and *checkBalance* refine the abstract event *withdraw* and *checkBalance* respectively. The transitions *insertCard*, *ejectCard* and *withdrawFail* are new events.

Fig. 9(c) shows the properties of the *withdrawOK* transition with the parameters, witness, guards and action. The witness specifies that the parameter *ac* represents the *self* 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 $(selfATM \in dom(atm\_card))$ and the card in the ATM is a valid card for the account whose balance is being modified $(selfATM.atm\_card = c$ and $c.card\_account = ac)$. Fig. 9(d) 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



(a) Refined state machine ATM_SM of refined class ATM of ATM_R2

(b) State machine active_atm_SM of refined state machine ATM_SM

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

achieved by elaborating the *active_atm* state into sub-states. The class diagram of machine **ATM_R2** contains three refined classes refining the classes Account, ATM and Card of **ATM_R1**. An attribute *atm_cash*, which represents the amount of cash stored in an ATM is added to the refined class ATM. A new class Pin is introduced which represents a set of ATM PIN numbers. The refined class Card has an association *card_pin* with the class Pin.The refined class ATM contains refined state machine ATM_SM which contains two refined states refining the states *idle* and *active_atm* of **ATM_R1** (Fig. 10(a)). The transitions *ejectCard1*, *ejectCard2*, *ejectCard3* and *ejectCard4* refine the abstract transition *ejectCard*. The transitions *insertCard*, *withdrawOK*, *withdrawFail* and *checkBalance* refine their corresponding abstract transitions in **ATM_R1**.

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*, *performedTrans* and *endTrans* (Fig. 10(b)). The state machine has a transition *insertCard* which elaborates the incoming transition to the refined super-state active_atm. The outgoing transitions *ejectCard1*, *ejectCard2*, *ejectCard3* and *ejectCard4* from the states *invalidCard*, *transOption*, *performedBankTrans* and *endTrans* respectively elaborate the outgoing transitions of the refined super-state *active_atm*. The transitions *withdrawOK*, *withdrawFail* and *checkBalance* elaborate the self loop transitions of the refined super-state *active_atm*. The transitions *validateCardOK*, *validateCardFail*, *withdrawATM* and *checkBalATM* are new transitions.

The third refinement models the request and response communication between the ATMs and the bank. The fourth refinement models the send and receive events of the request and response communication between ATMs and the bank. In third and fourth refinement, nested state machines are added to the ATM state machine forming four levels of nested state state machines. The fifth refinement introduces a form of communication between ATMs and the bank using message passing via two channels per ATM. The sixth refinement merged the channel pairs into single channel.

The state machine refinement in the second, third and fourth refinements introduced additional levels in the state machine nesting hierarchy. This supports a form of modular reasoning, since refinement invariants are only required for the states that are being elaborated, so it localizes proof effort.

All the models for the ATM development were constructed using the UML-B tool and corresponding Event-B machines were generated. All the proof obligations (POs) for the seven machines were generated and proved using the Rodin tool provers [6]. The total number of proof obligations (POs) is 962 in which all of them are proved automatically. The POs for each machine are: **ATM_A**:5, **ATM_R1**:35, **ATM_R2**:169, **ATM_R3**:156, **ATM_R4**:186, **ATM_R5**:358 and **ATM_R6**:53.

7 Conclusions

We have introduced notions of refined class and refined state machine for UML-B. We used these to described the following five refinement techniques:

- Add new attributes and associations to a refined class
- Add new classes in a refinement
- State elaboration
- Transition elaboration
- Move event to a refined class or a new class in a refinement

We extended the UML-B tool to support these new techniques.

UML-B enhances classical UML-B [12] which is a profile of UML that defines a subset and specialisation of UML. Classical UML-B is based on classical B rather than Event-B and it has restricted support for refinement. Some of the techniques used here (state elaboration, transition elaboration) were previously introduced by Snook and Walden [13] for classical UML-B. However, we provde a more precise definition of refined state machine and we provide tool support based on UML-B giving a different modeling visualization from the UML diagram symbols used in [13]. We also introduce class refinement techniques, which are not with dealt in [13]. In [14], a process for refinement involving the application of patterns that are based on the techniques introduced in [13] is suggested.

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 [16] 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. Also, the technique of state elaboration has been introduced in a refinement of UML state diagram [15] again without a formal notion of refinement. Simons [21] has presented a theory of compatible object refinement based on several proposed state-chart refinements that includes state elaboration.

There is much 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 [17] present the translation of UML-RSDS into classical B. The constraint language used is OCL whereas we use μB. Idani, Ledru and Bert [18] 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 modelling the communication between UML state charts in B in [23]. Other work on the integration of UML and B are in [22, 24] as outlined in [12]. Integration work of UML with Z has been investigated in [20]. In this work, class diagrams, state machines and the UML-RT structure diagrams are translated to CSP-OZ (an integrated formal method) specifications. In [19], a framework called UML + Z for building, analysing and refining models based on UML and Z is introduced. In [25], a transformation rules from VDM++ into UML class diagram and sequence diagram have been investigated.

We have presented the use of the above listed techniques in the ATM case study which was modelled using the UML-B tool. The Rodin tool was used to generate and prove the proof obligations. The approach of elaborating states with sub-states in refinement supports an incremental refinement approach. The hierarchical structure of nested state machines also supports modular reasoning by localising the invariants required for refinement proofs into the relevant state and its substates. An archive of UML-B development for the ATM case study can be uploaded [1]. Currently, we are working on the extensions to UML-B to support decomposition. We believe that the result of our research will be a methodology of refinement in UML-B which will assists modelling 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.(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)

[1] http://deploy-eprints.ecs.soton.ac.uk/95/

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, In: Proc. ICFEM, LNCS 4260, pp. 588-605. (2006)
7. Evans, N. and Butler, M. : A Proposal For Records in Event-B, In: Proc Formal Methods 2006, LNCS 4085, pp. 221-235, Springer-Verlag. (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. vol. 77, pp. 1-28, IOS Press. (2007).
10. Butler, M. and Yadav, D. : An Incremental Development of the Mondex System in Event-B, vol. 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, vol. 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 4355, pp. 171-185, Springer. (2006).
14. Plaska, M., Walden, M. and C. Snook: Documenting the Progress of the System Development. In: Proc. of Workshop on Methods, Models and Tools for Fault Tolerance. (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. and Vilbig, A.: Structuring and Refinement of Class Diagrams. In: Proc. of the 32nd Annual Hawaii International Conference, Vol. Track 6. (1999)
17. Lano, K., Clark, D. and Androutsopoulos, K.: UML to B: Formal Verification of Object Oriented Models. In: Proc. IFM2004, pp 761-768, Springer. (2004)
18. Idani, A. Ledru, L. and Bert, D.: Derivation of UML Class Diagrams as Static Views of Formal B Developments, In: Proc. ICFEM, pp. 37-51. (2005)
19. Amálio, N., Polack, F. and Stepney, S. : UML + Z: Augmenting UML with Z, In Marc Frappier, Henri Habrias, eds. Software Specification Methods : an Overview Using a Case Study, new edn, Hermes Science Publishing. (2006)
20. M. Moller, E. Olderog, H. Rasch and H. Wehrheim,: Linking CSP-OZ with UML and Java: A Case Study. In: Proc. IFM, pp. 267-286, Springer. (2004)
21. Simons, A. J. H.: A Theory of Regression Testing for Behaviourally Compatible Object Types: Research Articles, Journal Softw. Test. Verif. Reliab., vol. 16, no. 3, pp. 133-156, John Wiley and Sons Ltd. (2006)
22. Facon, P., Laleau, R., and Nguyen, H.P.: Mapping Object Diagrams into B Specifications. In Methods Integration Workshop, Electronic Workshops in Computing (eWiC), Springer-Verlag. (1996)
23. Ledang, H. and Souquiéres, J. : Contributions for Modelling UML State-Charts in B. In: Proc. IFM, LNCS 2335, pp. 109-127, Springer-Verlag. (2002)
24. Sekerinski, E. Graphical Design of Reactive systems. In D.Bert, editor, B'98: Recent Advances in the Development and Use of the B Method, LNCS 1393, Springer-Verlag, pp.182-197. (1998)
25. Lausdahl, K. G., Lintrup, H. K. A. and Larsen, P. G.: Coupling Overture to MDA and UML. Master Thesis. (2008)