TOWARDS REVERSE-ENGINEERING OF UML VIEWS FROM STRUCTURED FORMAL DEVELOPMENTS

Akram Idani and Bernard Coulette

University of Toulouse 2, IRIT, France
5 allée Antonio Machado, 31058 Toulouse Cedex 9, France

Keywords: B method, UML, Method integration, Meta-modelling, Reverse-engineering.

Abstract: Formal methods, such as B, were elaborated in order to ensure a high level of precision and coherence. Their major advantage is that they are based on mathematics, which allow, on the one hand, to neutralize risks of ambiguity and uncertainty, and on the other hand, to guarantee the conformance of a specification and its realization. However, these methods use specific notations and concepts which often generate a weak readability and a difficulty of integration in the development and the certification processes. In order to overcome this shortcoming several research works have proposed to bridge the gap between formal developments and alternate UML models which are more intuitive and readable. In this paper we are interested by the B method, which is a formal method used to model systems and check their correction by refinements. Existing works which tried to combine UML and B notations don't deal with the composition aspects of formal models. This limitation upsets their use for large scale specifications, such as those of information systems, because such specifications are often developed by structured modules. This paper improves the state of the art by proposing an evolutive MDA-based framework for reverse-engineering of UML static diagrams from B specifications built by composing abstract machines.

1 INTRODUCTION

The growing complexity of information systems is increasingly obvious and the control of risks inherent to their use becomes imperative and needs rigour and precision during their development. However, developing safe and secure information systems is difficult and error-prone. This motivates a significant amount of successful research to propose reliable investigation methods and techniques, based on mathematical foundations for secure systems development.

The success of software development in the case of METEOR (Behm et al., 1999) and the B method (Abrial, 1996) or the analysis of source code of Ariane 502, have shown that formal approaches give effective responses to these questions about safety and security. Indeed, formal methods, such as B, were elaborated in order to ensure a high level of precision and coherence. Their major advantage is that they are based on mathematics, which allow, on the one hand, to neutralize risks of ambiguity and uncertainty, and on the other hand, to realize software systems conform to their specifications. Still, these methods use specific notations and concepts which require a great knowledge of logic. Therefore, they remain dedicated specifically to secure and safe parts of Information Systems. Their mathematical notations often generate a weak readability and a difficulty of integration in the development and the certification processes. There is thus a risk that human errors such as misinterpretation of the requirements and specification documents lead to erroneously validate the specification, and hence to produce the wrong system. In order to fill these gaps, several research works have suggested to establish links between formal methods and UML. They have been focused on extending UML tool support to provide rigour via a partly-invisible and machine-generated formalisation.

In this context, several approaches (Sekerinski, 1998; Lano et al., 2004; Snook and Butler, 2006; Laleau and Polack, 2002) proposed rule-based techniques for deriving a formal B specification of an information system from UML models. However, they usually result in B models that are hard to read and quite unnatural. Consequently, as soon as resulting formal specifications are corrected, in case of incon-

Idani A. and Coulette B. (2008).
TOWARDS REVERSE-ENGINEERING OF UML VIEWS FROM STRUCTURED FORMAL DEVELOPMENTS.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - ISAS*, pages 94-103
DOI: 10.5220/0001697700940103

sistency, or refined in order to take into account system's constraints which are not specified in UML, the reverse translation is not guaranteed. The traceability of B model elements back to the original UML model becomes a serious problem, and failures to check proof obligations can be difficult to understand. In order to address this traceability challenge, we consider in this paper the inverse problem and propose to construct UML views from B specifications.

2 MOTIVATIONS

In previous works (Idani et al., 2007; Idani, 2006) we investigated a practical solution to assist the reverse-engineering of UML diagrams from B specifications: translations between B and UML notations are explicitly formalized in an evolutive MDA-based tool in terms of mappings between a proposed B meta-model and the UML meta-model. Our B-to-UML meta-model projections provide a clear framework which makes it easy to know on what semantic basis the transformation has taken place and hence remedy the lack of conceptual basis of existing translation approaches (Tatibouet et al., 2002; Fekih et al., 2006). Moreover, using our technique, we were able to scale up from small B specifications (several dozens of lines) to medium size ones[1] (several hundreds or thousand lines). Today, the largest B specification (*i.e.* the METEOR subway) is about 100,000 lines. In order to be able to scale up to such realistic sizes, an interesting new problem must be inevitably pointed up: the tool deals only with specifications built by a unique B abstract machine. However, one of the major reasons of the success of B in large projects (Behm et al., 1999) is the facility to build structured developments by composing several B machines. In the current version of the tool, as far as refinements and composition clauses (INCLUDES, SEES,etc) are concerned, developments are manually flattened into a single B machine. Our intention in this paper is then to improve the state of the art by evolving our contribution on this topic to address the reverse-engineering of UML static diagrams from compositional aspects in the B method.

This paper is organized as follows: Sect. 3 presents a brief overview of the B method and a case study. In Sect. 4 we propose an extension of our B meta-model to address compositional aspects in the B method. In Sect. 5 we discuss possible translations of the concept of abstract machine. Sect. 6 gives a set of rules for translating links between B abstract machine. In Sect. 7 we present the application of our rules. Finally, Sect. 8 draws the conclusions and perspectives of this work.

[1]Specifications successfully addressed by our B/UML tool are discussed in (Idani et al., 2005). For example: book store, travel agency, secure flight.

3 THE B METHOD

3.1 A Brief Overview

A software development process following the B method can be summarized by figure 1:

1. B specifications are written from a requirements document or from a detailed analysis of the requirements. The consistency of these formal specifications is ensured with the support of a proof tool.
2. These specifications follow a development process based on proved refinements that lead to executable programs.

An abstract machine is composed by basic clauses: MACHINE, SETS, VARIABLES, CONSTANTS, PROPERTIES, INVARIANT, INITIALISATION and OPERATIONS. The invariant properties are properties over variables which must always hold. Operations are the services of the abstract machine, they modify encapsulated variables using the generalized substitution principle.

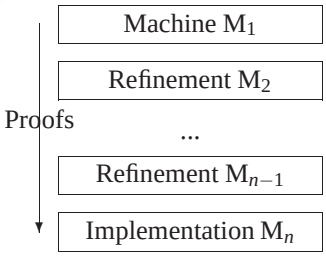

Figure 1: An incremental B development process.

3.2 A Simple Example

The example we consider in this section is inspired by (Abrial, 1999) and consists of two abstract machines: machine *ConferenceRoomGate* (Fig. 2) and machine *SecureConferenceAccess* (Fig. 3). This specification is that of a secure building containing a central hall and several conference rooms. In this model, we consider persons and objects that they carry, and we are interested mainly by: (*i*) opening and closing a conference room, (*ii*) the entrance of persons in the central hall of the building, and (*iii*) the access to a conference room.

MACHINE *ConferenceRoomGate*
SETS
GateStatus = {*open*, *closed*}
VARIABLES
state
INVARIANT
state ∈ *GateStatus*
INITIALISATION
state := *open*
OPERATIONS
open_gate = **PRE** *state* = *closed*
THEN *state* := *open*
END;
close_gate = **PRE** *state* = *open*
THEN *state* := *closed*
END
END

Figure 2: Machine *ConferenceRoomGate*.

Machine *ConferenceRoomGate* models the door of a conference room. In this abstract machine the gate states are listed in the enumerated set *GateStatus*. Opening and closing the door is realized by operations *open_gate* and *close_gate*. Variable *state* represents the current state of the door and takes its values from set *GateStatus*.

Machine *SecureConferenceAccess* is the specification of the access process to the central hall of the building and also the access to a conference room. The door of this conference room is designed by instance *ThisRoomGate* of machine *ConferenceRoomGate*. Abstract sets *Person* and *Object* represent respectively persons who want to accede to a conference room, and objects they carry. Constant *weared_objects* gives all the objects held by each person. Constant *Unauthorized_object* means all unauthorized material in the conference room (*e.g.* weapon, etc). Finally, variables *In_central_hall* and *In_conference_room* indicate positions of a person: either in the building's central hall, or in a conference room. The two operations of machine *SecureConferenceAccess* allow respectively the entrance of a person to the central hall and his entrance to the conference room. Note that other operations may be considered such as the screening check of each person, but for clarity of illustration and place reasons we present only a fragment of this model. Invariant of machine *SecureConferenceAccess* means that:

- a person can't be in both the central hall and the conference room,
- objects carried in the conference room are authorized in the conference room,
- the conference room gate remains open if the conference room is not empty, and as long as there are persons who have not yet attained the conference room.

MACHINE
SecureConferenceAccess
INCLUDES
ThisRoomGate.ConferenceRoomGate
SETS
Person ; *Object*
CONSTANTS
Unauthorized_object, *weared_objects*
PROPERTIES
Unauthorized_object ⊆ *Object* ∧
weared_objects ∈ *Object* → *Person*
VARIABLES
In_central_hall, *In_conference_room*
INVARIANT
In_central_hall ⊆ *Person* ∧
In_conference_room ⊆ *Person* ∧
(*In_central_hall* ≠ ∅ ∧ *In_conference_room* ≠ ∅
⇒ *ThisRoomGate.state* = *open*) ∧
∀ *pp*.(*pp* ∈ *In_conference_room*
⇒ *weared_objects*$^{-1}$[{*pp*}] ∩ *Unauthorized_object* = ∅)
INITIALISATION
In_central_hall, *In_conference_room* := ∅ , ∅
OPERATIONS
enter_central_hall =
PRE *ThisRoomGate.state* = *open* **THEN**
ANY *pp* **WHERE**
pp ∈ *Person* ∧
pp ∉ (*In_central_hall* ∩ *In_conference_room*)
THEN
In_central_hall := *In_central_hall* ∪ {*pp*}
END
END;
enter_conference_room =
PRE *ThisRoomGate.state* = *open* **THEN**
ANY *pp* **WHERE**
pp ∈ *In_central_hall* ∧
weared_objects^{-1}[{*pp*}] ∩ *Unauthorized_object* = ∅
THEN
In_conference_room :=
In_conference_room ∪ {*pp*} ||
In_central_hall := *In_central_hall* - {*pp*}
END
END
END

Figure 3: Machine *SecureConferenceAccess*.

4 A META-MODEL FOR THE COMPOSITIONAL ASPECTS

In this paper, our intention is not to present a complete B meta-model. We rather present an extension of a proposed meta-model (Idani, 2006) to focus on our B-to-UML translation rules.

Fig. 4 presents our B meta-model describing the abstract syntax of commonly used compositional aspects in the B method. We consider here only INCLUDES and REFINES links. The other composition mechanisms (*e.g.* USES, SEES, etc) can be seen

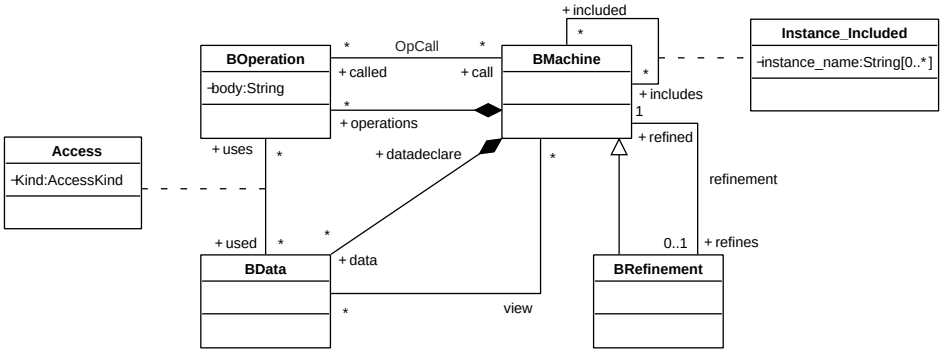

Figure 4: A meta-model for refinement and composition B structures.

as specializations of the INCLUDES relationship and they are naturally represented by a reflexive association similar to association *Instance_Included*.

In this meta-model, the meta-class *BMachine* represents the basic entity in the B method which is the abstract machine. We consider that a *BMachine* is composed by a set of operations (*BOperation*) and data (*BData*). Other constituents of a B machine (invariants, properties, etc) were discussed in (Idani et al., 2007). Meta-class *BData* concerns B variables and constants. A *BOperation* in a *BMachine* uses B data declared in the same machine following several kinds of access (Pre-condition, Reading, Writing, etc). This is defined by the associative meta-class *Access*.

In B, a refinement is a B machine which refines another B machine. Thus, meta-class *BRefinement* inherits from *BMachine* and it is linked to meta-class *BMachine* by association *refinement* (in order to distinguish refined and refining B machines).

Associative meta-class *Instance_icluded* represents the inclusion mechanism in the B method. Roles **`+includes`** and **`+included`** concern respectively including and included machines. Attribute *instance_name*[2] is given when an including machine includes one (or several) specific instance(s) of an included machine (*e.g.* instance *ThisRoomGate* of machine *SecureConferenceAccess*).

A B machine can refer (association *OpCall*) in its body (in clauses INITIALISATION and OPERATIONS) operations of machines that it includes. Still, association *OpCall* between a *BOperation* o and a *BMachine* $\mathcal{M}$ is possible only if an association *Instance_Included* or *refinement* exists between $\mathcal{M}$ and the *BMachine* in which o is defined.

Association *view* defines a data referencing from an including *BMachine* to the included one. The use of data (relation *view*) is specified by the visibility rules of the B theory. Indeed, an including machine can access data of the machine that it includes only in reading or through the operations of the included machine. Thus, specification of B compositions at a meta-level is rather focused on constraints specific to meta-classes *BMachine*, *BOperation* and *BData*. For example, a B machine references data or operations from other machines only if an inclusion or a refinement link is established.

5 POSSIBLE TRANSLATIONS OF META-CLASS *BMACHINE*

Abstract machine is the basic structuring concept of a B model, its goal is to gather coherently static (B data) and dynamic (B operations) elements. The decomposition of specifications in several abstract machines is based on purely logical criteria. In a UML view, such a structuring is realized by two possible mechanisms: classes and packages. Thus, we consider these two point of views when translating the meta-class *BMachine*. The first point of view (*BMachine* to *Class*) allows to see a B machine as a UML class which encapsulates attributes (B data) and methods (B operations). In the second point of view (*BMachine* to *Package*), a B machine is seen as a rather complex entity, whose constituents (data and operations) can be translated into well structured model elements (classes, associations, inheritance, etc). Fig. 5 gives a possible translation of machine *ConferenceRoomGate* in which both class and package point of views are combined.

Note that derivation of class attributes and methods, stereotypes and relations between the produced classes were discussed in (Idani et al., 2005). Thus, this paper doesn't discuss all derivation rules from B to UML, but it is focused on translations of composition links between B machines. We won't prescribe a mechanic algorithm for deriving structural views

[2]Multiplicity "[0..*]" of attribute *instance_name* shows that in an including machine several instance names can be used for a same B machine, and this name is not mandatory.

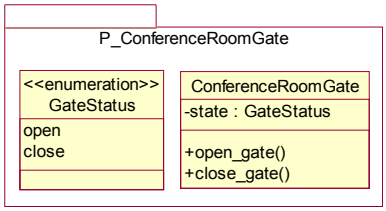

Figure 5: Translation of machine *ConferenceRoomGate*.

from B specifications. The transformation process described here is guided by heuristic rules depending on the appreciation of the analyst. Hence, in order to treat translations of compositional aspects in the B method, we propose the following main rules:

Rule 1. (*BMachine translation*)

Parameters:	$\mathcal{M}$: *BMachine*
Transformation:	$\mathcal{M}$ is translated into a package gathering a set of model elements. These packageable elements come from data (*BData*) and operations (*BOperation*) defined in $\mathcal{M}$.

Rule 2. (*BMachine translation*)

Parameters:	$\mathcal{M}$: *BMachine*
Transformation:	$\mathcal{M}$ is translated into a UML class. The structural and behavioral features of this class correspond to the data (*BData*) and operations (*BOperation*) defined in $\mathcal{M}$.

6 TRANSLATION OF COMPOSITIONAL ASPECTS

Since a B machine leads to a class or/and a package according to the selected translation rule, translation of composition relations (inclusion, refinement, etc) between B machines depends on possible dependencies between these resulting entities.

6.1 UML Packages Dependencies

In (Fig. 6) we show a fragment of the UML meta-model concerned with UML packages. UML packages dependencies are defined as referencing links which allow a source package to accede to elements (said public or visible) of a target package. These links are specified by meta-classes *ElementImport* and *PackageImport* and indicate a private or public visibility of the referenced elements (Fig. 6). While meta-class *ElementImport* lists elements of the target package which are specifically referenced by the source package, meta-class *PackageImport* covers all elements of the target package. In both cases, only members of the target package which have a public visibility can be referenced.

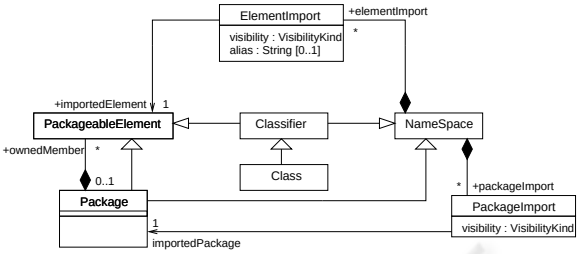

Figure 6: UML meta-model for package dependencies.

Basically, UML gives two kinds of package dependencies:

- Dependency ≪import≫: indicates a transitive package dependency; indeed, the import referencing of several elements of a package $\mathcal{P}_1$ by a package $\mathcal{P}_2$ is set for a package $\mathcal{P}_3$ which imports $\mathcal{P}_2$.
- Dependency ≪access≫: means that the package dependency is not transitive, and the scope of referenced elements in this case is limited to the source package of the access referencing.

The portion of UML meta-model given in Fig. 6 allows to distinguish dependencies between classes and packages. A package is a *NameSpace* which can import (`+elementImport`) a *PackageableElement*. This one may be a class (`+importedElement`) or a package. Furthermore, a class is a *NameSpace* too which can then import (`+packageImport`) a package (`+importedPackage`) or a class (`+importedElement`).

6.2 UML Classes Dependencies

Dependencies between UML classes can exist for several reasons (Fowler, 2004): one class sends a message to another class; some structural features of a class are stored in another class, one class uses another as a parameter type in one or several of its methods, etc. In this work, we consider three main class dependencies: (*i*) associations (allow to connect classes), (*ii*) call of methods (designed by the stereotype ≪call≫), and (*iii*) use of attributes (specified by stereotype ≪use≫).

6.3 Translation Rules

Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two B abstract machines (instances of meta-class *BMachine*). We assume that a composition link exists from $\mathcal{M}_1$ to $\mathcal{M}_2$. In the following, rule 3 and its sub-rules are proposed in

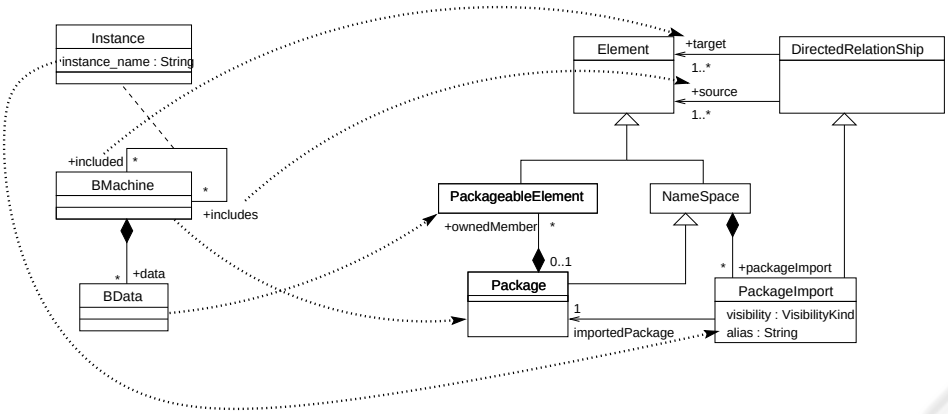

Figure 7: Illustration of rule 3.1.

case of an inclusion link (*i.e* we consider that $\mathcal{M}_1$ includes $\mathcal{M}_2$).

Rule 3. (INCLUDES *clause*)	
Parameters:	$\mathcal{M}_1, \mathcal{M}_2 : BMachine$
Precondition:	$\mathcal{M}_1$ includes $\mathcal{M}_2$
Transformation:	Execute one of the following sub-rules.

Considering that for each B machine, rules 1 and 2 (proposed in Sect. 5) produce either a class or a package, then several configurations must be taken into account.

6.3.1 Each BMachine Produces a UML Package

Rule 3.1

Precondition: $\mathcal{M}_1 : BMachine \xrightarrow{Rule_1} \mathcal{P}_1 : Package$ and $\mathcal{M}_2 : BMachine \xrightarrow{Rule_1} \mathcal{P}_2 : Package$

Transformation: The inclusion between $\mathcal{M}_1$ and $\mathcal{M}_2$ is translated into a dependance link from $\mathcal{P}_1$ to $\mathcal{P}_2$ with the stereotype ≪import≫. The alias name associated to this import dependance is that of the instance name of $\mathcal{M}_2$ in $\mathcal{M}_1$.

As the INCLUDES clause provides a transitive visibility mechanism then dependency between resulting packages is of type ≪import≫. Fig. 7 highlights projections from the proposed B meta-model to the UML meta-model when including and included abstract machines are translated into UML packages (*e.g.* $\mathcal{P}_1$ and $\mathcal{P}_2$). In this case we assume that instances of *BData* associated to $\mathcal{M}_1$ and $\mathcal{M}_2$ are translated into packageable model elements packed up in $\mathcal{P}_1$ and $\mathcal{P}_2$, and such that elements of $\mathcal{P}_2$ are used (or referenced) by elements of $\mathcal{P}_1$.

Rule 3.1. creates an instance of *PackageImport* to translate the inclusion mechanism, such that the source (**`+source`**) and the target (**`+target`**) of this instance are respectively $\mathcal{P}_1$ and $\mathcal{P}_2$. The instance name of the included machine used in the including machine (attribute *instance_name*), becomes a value of attribute *alias* in the instance of *PackageImport*. Finally, the attribute *visibility* of this instance of *PackageImport* is set to *public*, which corresponds to stereotype ≪import≫ (Fig. 8).

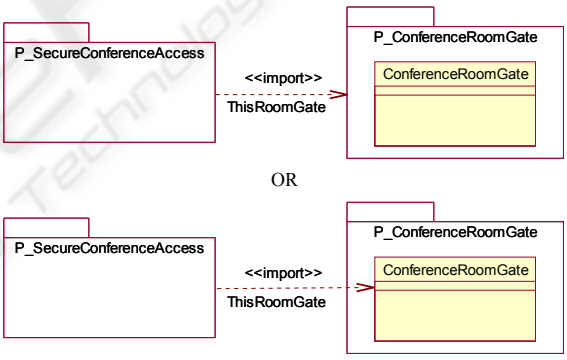

Figure 8: Possible applications of Rule 3.1.

Dependency ≪import≫ identified by Rule 3.1. may cover only some elements of $\mathcal{P}_2$, particularly those used in the body of $\mathcal{M}_1$. This is recommended when package $\mathcal{P}_2$ is quite complex. In this case, the inclusion mechanism doesn't lead to an instance of meta-class *PackageImport*, but rather to an instance of meta-class *ElementImport*. The source model element of this instance is the UML package produced from the including machine while its target is a packaged model element. For example, suppose that machine *ConferenceRoomGate* is translated into a package named *P_ConferenceRoomGate* and containing class *ConferenceRoomGate*, the resulting diagrams are illustrated in Fig. 8.

Considering that translation of the INCLUDES clause into a dependency ≪import≫ is justified by

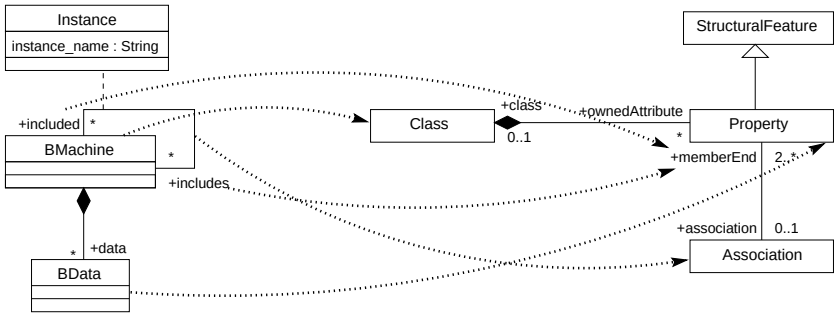

Figure 9: Illustration of rule 3.2.

(*i*) the visibility of model elements of the target package and (*ii*) the transitive character of this relationship, then we can consider similar translations of other B composition clauses. For example, the non-transitivity of clauses USES and SEES can be reduced to a dependency ≪access≫ between resulting packages.

6.3.2 Each BMachine Produces a UML Class

Rule 3.2

Precondition: $\mathcal{M}_1 : BMachine \xrightarrow{Rule_2} \mathcal{C}_1 : Class$ and $\mathcal{M}_2 : BMachine \xrightarrow{Rule_2} \mathcal{C}_2 : Class$

Transformation: Attributes of $\mathcal{C}_2$ corresponding to *BData*s used in clauses PROPERTIES or INVARIANT of $\mathcal{M}_1$, or referenced in the body of operations of $\mathcal{M}_1$ become public attributes (thus they can be used by Class $\mathcal{C}_1$). The inclusion between $\mathcal{M}_1$ and $\mathcal{M}_2$ is translated into an unidirectional association from $\mathcal{C}_1$ to $\mathcal{C}_2$, such that multiplicities associated to $\mathcal{C}_1$ and $\mathcal{C}_2$ are respectively 0..1 and 1. If an instance name of M_2 is explicitly defined in $\mathcal{M}_1$ then it becomes a role name beside $\mathcal{C}_2$. Otherwise, this role name is that of class $\mathcal{C}_2$.

Fig. 9 highlights projections from the proposed B meta-model to the UML meta-model when including and included abstract machines ($\mathcal{M}_1$ and $\mathcal{M}_2$) are translated into UML classes (*e.g.* $\mathcal{C}_1$ and $\mathcal{C}_2$). In this case, we assume that instances of *BData* and *BOperation* issued from each machine are translated into attributes and methods of classes $\mathcal{C}_1$ and $\mathcal{C}_2$.

For example, if machines *SecureConferenceAccess* and *ConferenceRoomGate* are translated into UML classes, then application of Rule 3.2. builds diagram of Fig. 10. Attribute *state* of class *ConferenceRoomGate* is set to public because variable *state* of machine *ConferenceRoomGate* is referenced in the invariant of *SecureConferenceAccess*, and it is used by operations *enter_central_hall* and *enter_conference_room*. Methods of class *SecureConferenceAccess* can accede to attributes and methods of class *ConferenceRoomGate* whereas the reverse is not allowed. An instance of *SecureConferenceAccess* is necessarily linked to a unique *ConferenceRoomGate* qualified by the role name *ThisRoomGate*. Furthermore, a *ConferenceRoomGate* participates to role *ThisRoomGate* for at the most one instance of *SecureConferenceAccess*.

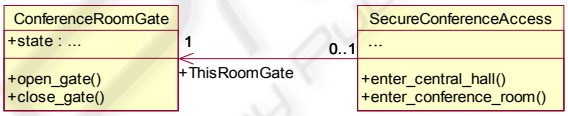

Figure 10: Application of Rule 3.2.

6.3.3 $\mathcal{M}_1$ is Translated into a Package and $\mathcal{M}_2$ is Translated into a Class

Rule 3.3

Precondition: $\mathcal{M}_1 : BMachine \xrightarrow{Rule_1} \mathcal{P} : Package$ and $\mathcal{M}_2 : BMachine \xrightarrow{Rule_2} \mathcal{C} : Class$

Transformation: Attributes of $\mathcal{C}$ corresponding to *BData*s used in clauses PROPERTIES or INVARIANT of $\mathcal{M}_1$, or referenced in the body of operations of $\mathcal{M}_1$ become public attributes. The machine inclusion is translated into a dependency ≪import≫ from $\mathcal{P}$ to $\mathcal{C}$. The alias name associated to this import dependancy is that of the instance name of $\mathcal{M}_2$ in $\mathcal{M}_1$. Classes of $\mathcal{P}$ are linked to class $\mathcal{C}$ by dependencies ≪use≫ or ≪call≫ according to a use of attributes or a call of operations of $\mathcal{C}$. Finally, stereotype ≪utility≫ is added to $\mathcal{C}$.

In this case we consider that on the one hand, *BData*s issued from $\mathcal{M}_1$ are translated into model elements gathered in package $\mathcal{P}$, and on the other hand, *BData*s issued from $\mathcal{M}_2$ are translated into attributes of a UML class $\mathcal{C}$.

Rule 3.3. builds several kinds of dependency between $\mathcal{P}$ and $\mathcal{C}$. In this translation, class $\mathcal{C}$ is a *utility* class to ensure that there is a unique set

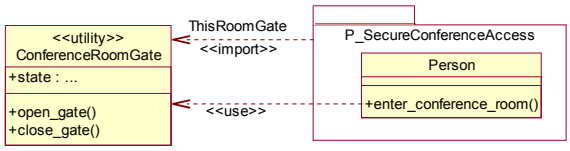

Figure 11: Application of Rule 3.3.

of data and services accessible by all classes of $\mathcal{P}$. Fig. 11 presents an example of application of Rule 3.3. In this diagram, we consider that the abstract set *Person* is translated into a UML class in package *P_SecureConferenceAccess* and that it encapsulates operation *enter_conference_room*.

Class *ConferenceRoomGate* acts as a module sharing global variables and procedures which are accessible by classes of package *P_SecureConferenceAccess*. In UML, this mechanism is represented by the stereotype ≪utility≫. The pre-condition access of variable *state* in operation *enter_conference_room* is translated into the dependency link ≪use≫.

Using the Design Pattern Singleton. The uniqueness of data and services showed by stereotype ≪utility≫ can be represented more precisely by the design pattern "Singleton" (Gamma et al., 1995). Indeed, it ensures that a class has only one instance and it provides a single point of access to this particular instance. We can then use the *Singleton* design pattern to allow classes of package $\mathcal{P}$ to share the single instance of $\mathcal{C}$. In the inclusion mechanism of the B method, operations of an including machine accede to specific instances of the included machine. For example, operations of *SecureConferenceAccess* have access to the instance *ThisRoomGate* of machine *ConferenceRoomGate*.

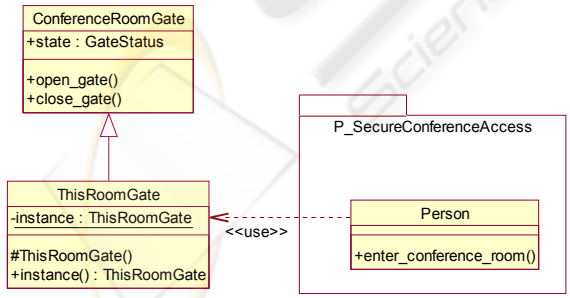

Figure 12: Application of design pattern "*Singleton*".

To illustrate the application of the design pattern *Singleton* on our case study we keep the translation of *SecureConferenceAccess* into a package containing the class *Person*. In the class diagram of Fig. 12, class *ThisRoomGate* is a specialization of class *ConferenceRoomGate*, as a result it inherits attribute *state* and methods *open_gate*() and *close_gate*(). Application of design pattern "*Singleton*" ensures that elements of package *P_SecureConferenceAccess* accede to a unique instance of *ThisRoomGate*.

Rule 3.4

Precondition: $\mathcal{M}_1 : BMachine \xrightarrow{Rule_1} \mathcal{P} : Package$ and $\mathcal{M}_2 : BMachine \xrightarrow{Rule_2} \mathcal{C} : Class$

Transformation: Attributes of $\mathcal{C}$ corresponding to *BData*s used in clauses PROPERTIES or INVARIANT of $\mathcal{M}_1$, or referenced in the body of operations of $\mathcal{M}_1$ become public attributes. The machine inclusion is translated into a class "*Singleton*" which is a sub-class of class $\mathcal{C}$. Classes of $\mathcal{P}$ are linked to class "*Singleton*" by dependencies ≪use≫ or ≪call≫ according to a use of attributes or a call of operations of $\mathcal{C}$.

6.3.4 $\mathcal{M}_1$ is Translated into a Class and $\mathcal{M}_2$ is Translated into a Package

In this case we consider that *BData*s issued from $\mathcal{M}_1$ are translated into attributes of a UML class $\mathcal{C}$, and those of $\mathcal{M}_2$ are translated into packaged model elements gathered in a package $\mathcal{P}$.

Rule 3.5

Precondition: $\mathcal{M}_1 : BMachine \xrightarrow{Rule_2} \mathcal{C} : Class$ and $\mathcal{M}_2 : BMachine \xrightarrow{Rule_1} \mathcal{P} : Package$

Transformation: The machine inclusion is translated into a dependency ≪import≫ from $\mathcal{C}$ to $\mathcal{P}$ such that instance name of $\mathcal{M}_2$ in $\mathcal{M}_1$ is used as an alias name.

Translation Rule 3.5 is similar to Rule 3.1. Dependency ≪import≫ produced by this rule allows to the UML class issued from $\mathcal{M}_1$ to accede to elements of the UML package issued from $\mathcal{M}_2$.

7 DISCUSSION

Rules proposed in this paper show that many translations are possible for the composition concept of the B method. They are then carried out interactively and may be applied simultaneously by the analyst in order to produce the most pertinent structural view. Each abstract machine which compose our case study can be translated into a class or a package depending on the application of rules 1 and 2. We believe that the choice amongst these two translation rules can be guided by the degree of detail and complexity of *BData*s issued from each B machine. For example, a

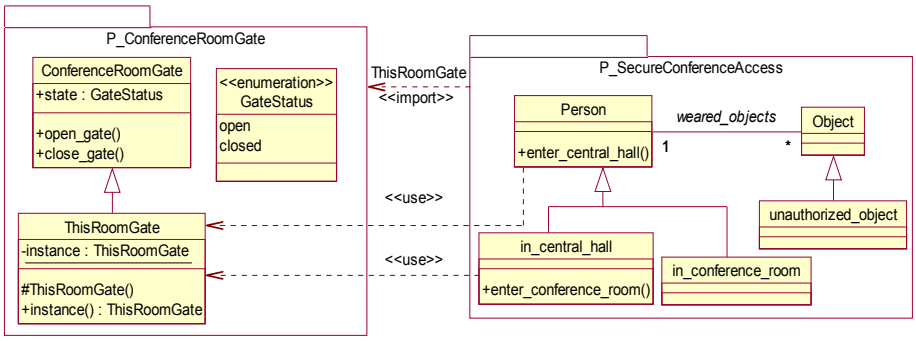

Figure 13: Application of the proposed rules on our case study.

B machine which consists of several abstract sets and relations between these sets can be viewed as a package gathering a set of classes and associations. Identification of the granularity degree depends on the way that the analyst specifies B models. Rigorous metrics and heuristics can be approached in order to help use the best rules. This opens a new interesting perspective to our contribution. However, this paper is intended to propose a palette of possible translations depending from these two main rules.

7.1 Application

Fig. 13 shows a possible application of our translation rules on the example of Sect. 3.2. In this diagram machine *SecureConferenceAccess* is translated into a UML package by applying rule 1, while machine *ConferenceRoomGate* is translated into a package and a class by applying simultaneously rules 1 and 2.

Package *P_SecureConferenceAccess*. It is a rather complex package gathering several classes and relations. Abstract sets are translated into classes, relations into associations, and set inclusion produces inheritance between classes. Operations *enter_central_hall* and *enter_conference_room* are encapsulated respectively by classes *Person* and *In_central_hall* for the following reasons: (*i*) *enter_central_hall* acts on persons to allow their access to the central hall of the building; and (*ii*) *enter_conference_room* concerns only persons who are in the central hall.

Package *P_ConferenceRoomGate*. In this package the enumerated set *GateStatus* becomes an ≪enumeration≫ class, variable *state* becomes an attribute of class *ConferenceRoomGate* and operations of machine *ConferenceRoomGate* are encapsulated by its corresponding class.

Links between Packages. In order to translate the INCLUDES link between machines of our example we considered the rules 3.1 and 3.4. Rule 3.1 is justified by the fact that both machines are translated into UML packages. As a result, a dependency ≪import≫ is created between these packages with the alias name *ThisRoomGate*. Rule 3.4 is applied because machine *ConferenceRoomGate* is translated into a UML class and also because machine inclusion in this case specifies a unique instance of *ConferenceRoomGate* called *ThisRoomGate*.

7.2 Existing Works and Contribution

Derivation of UML diagrams from B specifications could be built using two kinds of tools:

1. Tools that extract the static aspects of the B specifications. For example (Tatibouet et al., 2002) defined some rules to automatically derive a UML class diagram from formal B specifications. In the same context, (Fekih et al., 2006) presents some heuristics which lead to interactively construct simpler diagrams.
2. Tools that represent the behaviour of the specifications in form of state/transition diagrams. For example, (Bert and Cave, 2000) starts from a user-guided choice of significant states and applies proof techniques to explore all the valid transitions of the system. In (Leuschel and Butler, 2003) a model checker of B specifications is presented. This tool produces concrete state/transition diagrams where states are valuations of B variables and transitions are operation calls. Our contribution in this context (Idani and Ledru, 2006) combines animation and proof to produce abstract state/transition diagrams more intuitive and readable.

This paper studied the first kind of tools which extract a UML structural view from existing formal specifications. It is the first attempt for reverse-engineering of UML diagrams from composition clauses of the B method. Indeed, existing works including our previous contribution (Idani and Ledru, 2006; Idani et al., 2005) deal with specifications built by a unique B machine.

8 CONCLUSIONS

A crucial idea of Model Driven Engineering is that transformations between heterogeneous models can be described uniformly in terms of meta-model mappings. Based on the fact that meta-models define an abstract syntax from which one can describe model semantics, transformation rules that arise from MDA-based techniques are explicit and precise.

In our contribution we applied such a technique in order to address the reverse-engineering of UML structural diagrams from formal B specifications of information systems. Derivation rules are described in an MDA framework in terms of projections from a proposed B meta-model to the UML meta-model. Currently, we are working on integrating efficiently the proposed extensions to our MDA-based CASE tool for combining UML and B notations. The tool is intended, on the one hand, to circumvent the lack of traceability of UML-to-B approaches, and on the other hand, to provide a useful UML documentation of B developments in order to help understanding formal specifications. Still, further research is needed to broaden the scope of our method and its support tool and cover more B constructs (*e.g.* refinements, etc). Another interesting perspective is to integrate to our tool a checker which allows to apply transformations only if the B specifications are neither redundant nor inconsistent among them. The intention of such a B specifications checker is to avoid derivation of incorrect UML diagrams.

Bridging the gap between UML and B notations have received a lot of interest in the last decade from the scientific community. However existing approches were not applied on large-scale applications. Experiments done by our tool (Idani et al., 2005) (presented in Sect. 2) showed encouraging results in this direction, and the approach presented in this paper provides a better coverage of B and UML.

REFERENCES

Abrial, J.-R. (1996). *The B-book: assigning programs to meanings*. Cambridge University Press.

Abrial, J.-R. (1999). System study: Method and example. http://www-lsr.imag.fr/B/Documents/ClearSy-CaseStudies/.

Behm, P., Benoit, P., Faivre, A., and Meynadier, J.-M. (1999). METEOR: A successful application of B in a large project. In *FM'99*, volume 1709 of *LNCS*, pages 369–387. Springer-Verlag.

Bert, D. and Cave, F. (2000). Construction of Finite Labelled Transition Systems from B Abstract Systems. In *Integrated Formal Methods*, volume 1945 of *LNCS*, pages 235–254. Springer-Verlag.

Fekih, H., Jemni, L., and Merz, S. (2006). Transformation of B Specifications into UML Class Diagrams and State Machines. In *21st Annual ACM Symposium on Applied Computing*, pages 1840–1844. ACM.

Fowler, M. (2004). *UML 2.0*. CampusPress.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

Idani, A. (2006). *B/UML: Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B*. PhD thesis, Université de Grenoble - France.

Idani, A., Boulanger, J.-L., and Philippe, L. (2007). A generic process and its tool support towards combining UML and B for safety critical systems. In *20th Int. Conf. on Computer Applications in Industry and Engineering*, pages 185–192, USA.

Idani, A. and Ledru, Y. (2006). Dynamic Graphical UML Views from Formal B Specifications. *International Journal of Information and Software Technology*, 48(3):154–169. Elsevier.

Idani, A., Ledru, Y., and Bert, D. (2005). Derivation of UML Class Diagrams as Static Views of Formal B Developments. In *International Conference on Formal Engineering Methods (ICFEM'05)*, volume 3785 of *LNCS*, pages 37–51, UK. Springer-Verlag.

Laleau, R. and Polack, F. (2002). Coming and Going from UML to B: A Proposal to Support Traceability in Rigorous IS Development. In *B'02*, volume 2272 of *LNCS*, pages 517–534. Springer.

Lano, K., Clark, D., and Androutsopoulos, K. (2004). UML to B: Formal Verification of Object-Oriented Models. In *Integrated Formal Methods*, volume 2999 of *LNCS*, pages 187–206. Springer.

Leuschel, M. and Butler, M. (2003). ProB: A Model Checker for B. In *FME 2003: Formal Methods Europe*, volume 2805 of *LNCS*, pages 855–874. Springer-Verlag.

Sekerinski, E. (1998). Graphical Design of Reactive Systems. In *B'98*, pages 182–197, London, UK. Springer.

Snook, C. and Butler, M. (2006). UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122.

Tatibouet, B., Hammad, A., and Voisinet, J. (2002). From an abstract B specification to UML class diagrams. In *2nd IEEE International Symposium on Signal Processing and Information Technology*.