Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 133 (2005) 159–174

www.elsevier.com/locate/entcs

Object Oriented Concepts Identification from Formal B Specifications

Akram Idani, Yves Ledru[1]

Laboratoire Logiciels, Systèmes, Réseaux - IMAG
Université Joseph Fourier
Grenoble, France

Abstract

This paper addresses the graphical representation of static aspects of B specifications, using UML class diagrams. These diagrams can help understand the specification for stakeholders who are not familiar with the B method, such as customers or certification authorities. The paper first discusses some rules for a preliminary derivation of a class diagram. It then studies the consistency of the concepts preliminarily identified from an object oriented point of view. A formal concept analysis technique is used to distinguish between consistent classes, attributes, associations and operations. The proposed technique is to incrementally add operations to the formal specification which automatically result in evolution of the class diagram.

Keywords: B, UML, integrated methods.

1 Introduction

Formal methods are nowadays the most rigorous way to produce software. They provide techniques to ensure the consistency of a specification and to guarantee that some piece of code implements a given specification. Several industries involved in safety critical activities, like the railway industry, have perceived the benefits of such approaches and significant developments like the Paris Meteor subway have been partially performed using formal methods[1]. Still, while formal methods provide solutions to the verification problem ("do

[1] Email: {Akram.Idani, Yves.Ledru}@imag.fr

doi:10.1016/j.entcs.2004.08.063

the system right"), the validation problem ("do the right system") remains a major challenge for formal methods engineers.

One of the difficulties is to make sure that the various stakeholders of a critical project (developers, customers, certification authorities) agree on the meaning of the formal specification. Usually, there is a cultural gap between formal methods, with their mathematical concepts and notations, and the usual techniques of the various stakeholders in such a project (graphical formalisms and natural language documents). There is thus a significant risk that 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 bridge this gap, several research teams have proposed approaches to integrate formal and graphical specifications [14,15]. A possible integration strategy is to give graphical syntax to existing notations, e.g. in the early 90s Dick and Loubersac [5] proposed a graphical syntax for VDM. Another approach is to design industrial languages which integrate formal concepts. This approach underlies the development of several UML notations. For example, UML State Transition diagrams are deeply influenced by several state machine languages, and in particular Statecharts[9]. Also the Object Constraint Language [23] includes concepts from model-based specification languages such as Z[20].

A significant effort has been devoted by the research community in order to establish links between UML and formal methods. In particular, several approaches provide translations from annotated UML diagrams to formal methods [6,11,13,16,18,19]. These approaches aim to take advantage of formal methods tools while remaining integrated in a standard industrial process based on UML.

This paper investigates the reverse approach: using graphical notations, such as UML diagrams, as a way to document formal developments. It starts from the fact that several significant formal developments are mainly based on formal methods. For example, the B method has been used in industrial railway[1] and smart card[4] applications. Such critical applications must usually be accepted by independent certification authorities that are not necessarily expert in formal methods. Therefore, it makes sense to construct graphical views from the formal developments as an additional documentation. It is expected that these more intuitive representations will be easier to accept by certifiers.

The graphical representations could be build using two kind of tools:

(i) Tools that extract the static aspects of the B specifications. For example [21] defined some rules to automatically derive an UML class diagram

from formal B specifications. In the same context, [7] presents some heuristics which lead to construct interactively simpler diagrams.

(ii) Tools that represent the behaviour of the specifications. [2] is the first proposal for the derivation of a visual representation of the behaviour of a given B specification. This behaviour is described using finite Labelled Transition Systems (LTS). Starting from a user-guided choice of significant states, proof techniques are used to explore all the valid transitions of the system. The resulting LTS can then be further reused to verify temporal logic properties by model-checking. In the same context [22,8] proposed some derivation rules for the generation of statecharts from B specifications. Finally, in [17] a tool assisting the animation of B specifications is presented, it also helps in the construction of state transitions diagrams. In these diagrams states are valuations of B variables and transitions are some operation call.

This paper studies the first kind of tools which extract a structural view from existing formal specifications in order to ease their understanding and maintenance. The extraction of structural information is a classical problem in the software maintenance community. In fact, understanding how an application is organized is a major factor when maintaining it especially when its structure is complex and documentation is unavailable or outdated. Most of program comprehension techniques aim to give a modular restructuring of software which eases its maintenance. We believe that similar techniques can be applied to formal specifications.

This paper proposes a two-step approach to the construction of a class diagram from a B specification.

- The initial step (Sect. 2) applies generation rules to the B specification to translate B concepts into elements of a class diagram. It is close to existing approaches such as [21,7].
- While the first step is only concerned with the variables of the B specification, the second step (Sect. 3) also takes into account the operations. A concept dependance technique is used to help locate the appropriate class for each operation.

Section 4 further studies the robustness of the class diagram to evolutions of the B specification. Finally, Sect. 5 draws the conclusions and the perspectives of this work.

2 Generation of a preliminary class diagram

UML class diagrams and B share similar concepts such as the encapsulation of operations and variables, or the notion of association. This motivates several researches to translate one language into the other. This section will highlight how this overlap of concepts can be exploited to automatically translate B specifications into a UML class diagram.

The approaches traducing UML diagrams into B formal specifications give different rules for the traceability between graphical and formal concepts. For example, in [11,12] a class c is translated into a given set Sc and a variable set Vc and an invariant property which is: $Vc \subseteq Sc$.

As the derivation is made automatically, we think that it may be judicious to reason similarly and to derive an UML diagram from information given by the formal B specification. For example a given set Se in a formal B specification may be seen as an abstraction of a set of objects and may be translated as an UML class.

2.1 A simple example

The AccessControl specification is that of the control of the access of authorized users to the resources in a network. The corresponding B specification is given in the example 2.1.

Example 2.1

MACHINE *AccessControl*
SETS
USERS ; *RESOURCES* ; *ADDRESSES*
VARIABLES
Permitted, *Assigned* , *IpAdress*, *Unused*
INVARIANT
$Permitted \in USERS \leftrightarrow RESOURCES \wedge$
$IpAddress \in ADDRESSES \rightarrowtail RESOURCES \wedge$
$Assigned \in USERS \nrightarrow RESOURCES \wedge$
$Assigned \subseteq Permitted \wedge$
$Unused \subseteq RESOURCES \wedge$
$Unused \cap ran(Assigned) = \emptyset$
INITIALISATION
$Permitted, Assigned := \emptyset, \emptyset \;||$
$IpAddress, Unused := \emptyset, \emptyset$
OPERATIONS
Add_Permission(*user*, *resource*) =
PRE
$user \in USERS \wedge$
$resource \in RESOURCES \wedge$
$(user \mapsto resource) \notin Permitted$
THEN
$Permitted :=$
$Permitted \cup \{user \mapsto resource\}$
END

$st \leftarrow$ **List_USERS**(*resource*) =
PRE
$resource \in RESOURCES$
THEN
$st :=$ **dom**$(Permitted \rhd \{resource\})$
END;

Assign_Resource(*user*, *resource*) =
PRE
$user \in USERS \wedge$
$resource \in RESOURCES \wedge$
$(user \mapsto resource) \in Permitted \wedge$
$user \notin$ **dom**$(assigned)$
THEN
$Assigned :=$
$Assigned \cup \{user \mapsto resource\}$
END
END

This specification features three abstract sets which are USERS, RE-

SOURCES and ADDRESSES. The network resources are essentially computers identified by their IP addresses (variable *IpAddress*). Users must have a permission to access to a given resource (variable *Permitted*). A user can only use one resource at a time. The variable *Assigned* means that a user is using a resource. Finally, the subset of RESOURCES called *Unused* gives the set of unused resources. We consider three basic operations:

- Add_Permission(user,resource): gives the permission to a given user to access to a resource;
- List_USERS(resource): lists the users having the permission to access a given resource;
- Assign_Resource(user,resource): assigns a user to a resource for which he has permission.

2.2 Rules for a derivation of a class diagram

This section will propose some preliminary rules for the derivation of a class diagram directly from the variables and constants of the B specification. We will try to identify an analogy between B and object oriented concepts.

2.2.1 Classes

In the Object Oriented paradigm a class represents an abstraction of entities with common characteristics. It represents a set of concrete objects which are called instances. The formalization of the existence of objects can be done independently from the structure of objects themselves. It just requires a set of objects identities. This is typically the concept of sets in B which allow us to identify them as an abstract representation of some concrete elements.

Rule 1: Sets in the B specification correspond to classes in the UML specification.

For example, we identified for the machine of the example 2.1 three classes which are USERS and RESOURCES and ADDRESSES corresponding respectively to the abstract sets declared in the *SETS* clause.

2.2.2 Associations

An association is a bidirectional connection expressing a relationship between classes. It is an abstraction of the possible links existing between objects instances of classes. As we translated abstract sets into classes, associations will be every relation existing between these sets.

Rule 2: Relations in the B specification correspond to associations in the UML specification.

In Fig. 1, Permitted and Assigned appear as associations between *USERS* and *RESOURCES*.

In UML some constraints should be expressed on a relation or a group of relations. For example in the AccessControl machine invariant properties define a constraint between the two relations *Permitted* and *Assigned*. This constraint could be seen as the traditional *subset* constraint between two collections (also called roles).

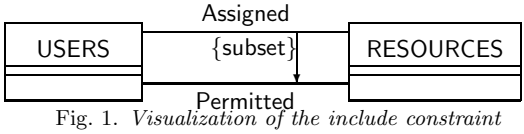

Fig. 1. *Visualization of the include constraint*

2.2.3 Roles

The extremity of an association defines a role which is a pseudo-attribute of the source class. In the B specifications there are no concepts being able

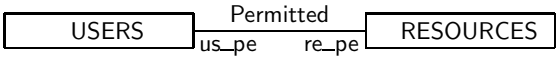

Fig. 2. *Identification of roles in a derived class diagram*

to be traceable into a role. We choose then to identify manually roles as a combination of the two first characters of the identified source class name and its association name.

2.2.4 Multiplicity

The multiplicity in UML is a constraint on the number of links which may exist between objects. Specializations of B relations correspond to various multiplicity constraints.

Rule 3: Multiplicities are derived from the B specification using the following table:

Relation type	Symbol	A	B	Relation type	symbol	A	B
Relation	$A \leftrightarrow B$	*	*	Partial surjection	$A \twoheadrightarrow B$	1..*	0..1
Partial	$A \nrightarrow B$	*	0..1	Total surjection	$A \rightarrow\!\!\!\!\!\rightarrow B$	1..*	1
Total	$A \rightarrow B$	*	1	Partial bijection	$A \rightarrowtail\!\!\!\!\!\!\rightarrow B$	1	0..1
Partial injection	$A \rightarrowtail B$	0..1	0..1	Total bijection	$A \rightarrowtail\!\!\!\!\!\!\rightarrow B$	1	1
Total injection	$A \rightarrowtail B$	0..1	1				

2.2.5 Inheritance

Having the equivalence between a set and a class, the subset concept can be translated into a subclass. Similarly with the class generalization mechanism,

the set inclusion defines a conceptual sub-collection of a set of objects.

Rule 4: a subset in the B specification is translated as a specialized class in the UML diagram.

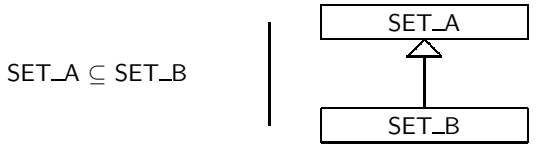

Fig. 3. *Formal and Object Oriented Concepts matching for sublasses*

2.3 Discussion

Applying the previous rules we obtain the diagram of fig. 4.

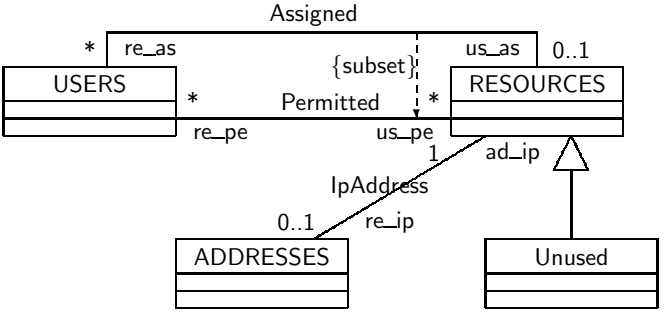

Fig. 4. *Preliminary derived class diagram for the AccessControl specification*

Although we believe that Fig. 4 gives a readable diagram for the Access-Control specification, we are afraid that the systematic application of the rules given in this section will not scale up for realistic B specifications. In particular, every set of values, including sets of basic types such as INTEGER or STRING result in corresponding classes.

For example, if the specification contains a string variable called *Name* and defined such that: Name ∈ USERS → STRING, we will obtain a structure expressed by fig.5.

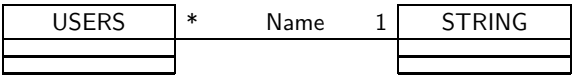

Fig. 5. *Basic class diagram representing data structure*

Intuitively *name* should be an attribute of the considered class. In order to distinguish between class candidates and attribute candidates, Sect. 3 will take into account the operations of the B specification that access these attributes or classes.

Still we believe that this first step gives comparable or better results than the existing approaches.

The rules given in [21] generate classes for every machine, set, and relation of the B specification. Moreover, these classes are linked by numerous associations. Only boolean and integer variables become attributes of these classes. This results in many more classes than our approach and hence in more complex diagrams.

In [7] classes are derived with respect to a simple rule which identifies the domain of a relation as a class if it corresponds to an abstract set. Applying this rule on the example 2.1 we obtain two classes corresponding to the sets *USERS* and *ADDRESSES* without association (Fig. 6). The relations *Permitted* and *Assigned* and *IpAddress* are then considered as attributes typed by their codomains *RESOURCES*. The empirical rules given by [7] lead to an incomplete class diagram which doesn't show unused resources because *RESOURCES* is not identified as a class.

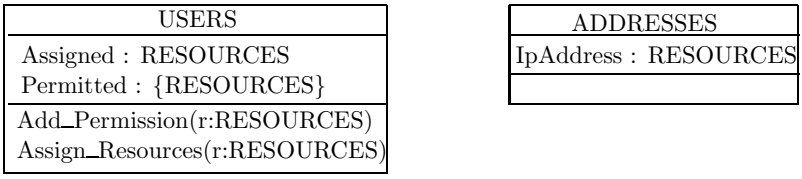

Fig. 6. *Class diagram for the AccessControl Machine generated using the rules of [7]*

3 Taking operations into account

The previous section has identified a set of class candidates. In this section, we will try to sort out these candidates into potential classes or attributes. In order to evaluate the pertinency of a class candidate, a concept dependance context will be built which relates the class candidates to the operations of the specification.

Our study is focused on the dependance between B operations and data identified in the preliminary class diagram. First, we study the relationship between B operations and classes and associations of the preliminary class diagram in order to determine their pertinence from an object oriented point of view. Then, we decide about the distribution of those elements over the pertinent class diagram entities. Finally, we add the other data of the B specification as attributes of the definitely identified classes.

We can informally define pertinency as follows:

- The pertinency of an operation vs a class measures if it makes sense to associate this operation as a method of the class.
- A class is pertinent if all its methods are pertinent. A class without methods has a low pertinency.

• An association is pertinent if it links pertinent classes.

Pertinency is thus defined on basic of operation pertinency. In order to measure operation pertinency, we use a concept dependance graph. Elements of the preliminary structure and operations form a context called concept dependance context. In the defined context we consider only concepts from the preliminary class diagram which are manipulated by the operations. Concepts correspond thus to either classes or associations.

The dependance between the B operations and the preliminary identified concepts is formalized as a bipartite graph G. This graph is a representation of a *concept dependance context* which is formally expressed as follows:

Definition 3.1 A **concept dependance context** $\mathcal{G} = (\mathcal{C},\mathcal{O},\mathcal{I})$ consists of two sets $\mathcal{C}$ and $\mathcal{O}$ and a binary relation $\mathcal{I}$ between $\mathcal{C}$ and $\mathcal{O}$ (with $\mathrm{dom}(\mathcal{I}) = \mathcal{C}$). The elements of $\mathcal{O}$ are the operations of the considered B specification and the elements of $\mathcal{C}$ are the preliminary identified concepts corresponding to some formal elements in the specification. A concept c ($c \in \mathcal{C}$) is in relation $\mathcal{I}$ with an operation o ($o \in \mathcal{O}$) if and only if o uses the formal data corresponding to c.

The definition of a concept dependance context requires a more precise definition of the "use" of a concept by an operation. In the B method we can identify three kinds of "uses":

(i) Modification: the operation accesses and modifies formal data corresponding to the concept

(ii) Reading: Performing the operation requires read access to the formal data corresponding to the concept

(iii) Precondition access: the precondition of the operation refers to the formal data corresponding to the concept.

We have chosen to treat only concepts which appear explicitly in the specification of operations. Therefore, "uses" is defined in our approach as an explicit reference to the considered concept. For example, the concept *USERS* is not treated in relation with the operation *List_USERS* because it doesn't appear explicitly in its body.

Definition 3.2 For a set $\mathcal{A} \subseteq \mathcal{C}$ of concepts we define:

$$\mathrm{Op}(\mathcal{A}) = \{o \mid o \in \mathcal{O} \wedge \forall c \cdot (c \in \mathcal{A} \Rightarrow (c,o) \in \mathcal{I})\}$$

The set of operations common to the concepts in $\mathcal{A}$.

Definition 3.3 For a concept dependance context $\mathcal{G}$ we define the maximal concept part such that:

$$c \in \max(\mathcal{G}) \Leftrightarrow \forall\, s \cdot (s \in \mathcal{C} - \{c\} \Rightarrow \mathrm{Op}(\{c\}) \not\subseteq \mathrm{Op}(\{s\}))$$

Where $\max(\mathcal{G})$ is the set of all maximal concept parts of $\mathcal{G}$.

Example 3.4 The concept dependance context $\mathcal{G}$ issued from the AccessControl specification can be represented by a conceptual bipartite graph. This graph simply means that there are no arcs between a concept and another concept, and no arcs between an operation and another operation. All arcs go from a concept to an operation. The considered graph is defined by the set of concepts $\mathcal{C}$={USERS, RESOURCES, Permitted, Assigned}, the set of operations $\mathcal{O}$={Add_Permission, List_USERS, Assign_Resource}, and a matching corresponding to the incidence function $\mathcal{I}$ given in Fig. 7. Although ADDRESSES, Unused and IpAddress were identified as preliminary concepts in Fig. 4, they don't appear in Fig 7 because they are not "used" by any of the operations. Please notice that the set of concepts includes both classes and associations.

In this example, RESOURCES and Permitted are maximal parts; $\max(\mathcal{G})$ is thus equal to {RESOURCES, Permitted}.

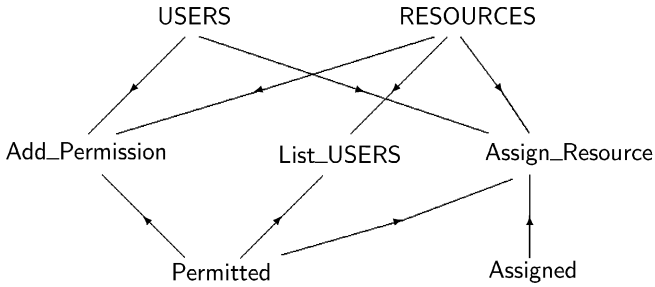

Fig. 7. *Bipartite graph of the concept dependance context issued from the AccessControl machine*

We can now define the notion of pertinency for a class or an association:

Definition 3.5 A class c ($c \in \mathcal{C}$) is pertinent iff $c \in \max(\mathcal{G})$.

Definition 3.6 An association a ($a \in \mathcal{C}$) between two classes c_1 and c_2 ($\{c_1,c_2\} \in \mathcal{C}$) is pertinent iff $\{c_1,c_2\} \in \max(\mathcal{G})$ and $\mathrm{Op}(a) \subseteq \mathrm{Op}(c_1) \cup \mathrm{Op}(c_2)$.

Pertinent concepts will become the classes and the associations of our class diagram. The remaining classes and associations which appeared on the preliminary class diagram and are not pertinent will become attributes of the pertinent classes.

Rule 5: The class diagram features all pertinent classes and associations.

Rule 6: A non pertinent class c_1 ($c_1 \in \mathcal{C}$) becomes an attribute or an attribute type in a class c_2 ($c_2 \in \mathcal{C}$) iff $c_2 \in \max(\mathcal{G})$ and $\mathrm{Op}(c_1) \subseteq \mathrm{Op}(c_2)$.

From the definition of non pertinent classes, we know that at least one c_2 exists. If several pertinent classes can have c_1 as an attribute, the choice is left to the analyst.

Similar rules are given for associations and operations:

Rule 7: A non pertinent association a ($a \in \mathcal{C}$) becomes an attribute in a class c ($c \in \mathcal{C}$) iff $c \in \max(\mathcal{G})$ and $\mathrm{Op}(a) \subseteq \mathrm{Op}(c)$. The identified attribute will then be typed by one extremity of the association a.

Rule 8: An operation o ($o \in \mathcal{O}$) becomes a method in a class c_1 ($c_1 \in \mathcal{C}$) iff $c_1 \in \max(\mathcal{G}) \wedge (o \in \mathrm{Op}(c_1) \vee (\exists\ c_2 \cdot (c_2 \in \mathcal{C} \wedge c_2 \in \mathrm{Attributes}(c_1) \wedge o \in \mathrm{Op}(c_2))))$

In our example, we have $\max(\mathcal{G}) = \{\mathit{RESOURCES},\ \mathit{Permitted}\}$. This set of maximal parts only includes a single class (RESOURCES). Following our rules, it becomes the only class of our class diagram (Fig. 8). The remaining concepts of Fig. 7 become attributes or types of this class. Since the three operations of the B specification use RESOURCES, they become methods of this class.

Although Permitted was a maximal part, it is not a pertinent association because it does not satisfy definition 3.6. Also, following rule 6, USERS can either become an attribute or a type. Since it is a constant set, we felt it should be modelled as a type.

Some concepts of the preliminary diagram (ADDRESSES, IpAddress, Unused) no longer appear in the resulting diagram. Actually, since our rules are based on usage of these concepts by operations, concepts which do not participate to the behaviour of the B machine do not appear in the resulting diagram.

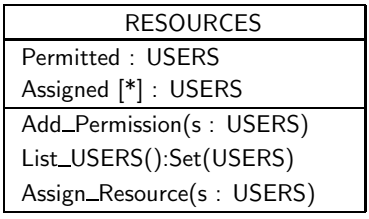

Fig. 8. *Pertinent class diagram from the AccessControl Machine*

Comparison with other approaches.

The approach of [21] systematically groups operations of the B machine in the class corresponding to the machine. In the section 4, we will see that our method does not always result in grouping all operations in a single class. We feel that this ability to share operations amongst several classes is definitely needed when B machines get larger.

Fig. 6 results from the applications of the rules of [7] to the AccessControl example. In fact, it is impossible from this class diagram to find the list of users of a resource (operation *List_USERS*). Such an operation could not be a method of the class *USERS* because it refers to several instances of this class. It must rather be a method of a class RESOURCES. Unfortunately, RESOURCES only appears as a type in fig. 6. The figure generated by applying our method (figure 8) is therefore more correct.

We feel thus that the second step of our approach, which takes operations into account to evaluate the pertinency of classes, results in improvements over existing methods.

4 Incremental development of the B specification and its impact on the B graphical representation

This section will investigate the robustness of the graphical representation when evolving the B specification. We introduce in the AccessControl machine an operation which gives the resources assigned to a user. The introduction of this operation into the previous context modifies the graph representation of concept dependance context as follows:

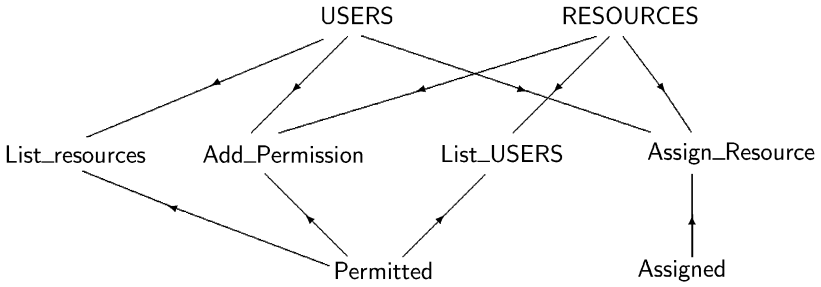

Fig. 9. *A new concept dependance context adding the operation List_Resources*

Having this new context, the maximal concept parts becomes equal to {*USERS*, *RESOURCES*, Permitted}. The two first concepts identified as preliminary classes are identified now as pertinent. The class which will encapsulate the shared operations (Add_Permission and Assign_Resource) can be either *USERS* or *RESOURCES*, we choose arbitrarily to put them in the class USERS. The relations Permitted and Assigned become two pertinent associations because Op(*Permitted*) $\subseteq$ Op(USERS) $\cup$ Op(RESOURCES) and Op(*Assigned*) $\subseteq$ Op(USERS) $\cup$ Op(RESOURCES). Fig. 10 shows the resulting class diagram.

Let us now add three other operations:
- Change_IP(ip,resource): changes the IP address of a resource;

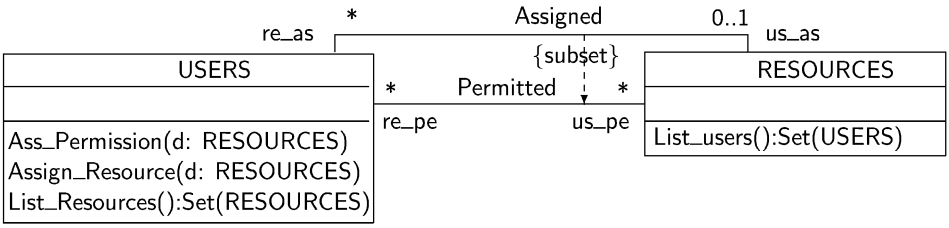

Fig. 10. *An evolved class diagram for the AccessControl specification*

- Deassign_Resource(resource): deassigns the resource and adds it to the unused set;
- Disconnect(unused): disconnects an unused resource by deleting its IpAddress.

These operations will change the representation of the new context as follows:

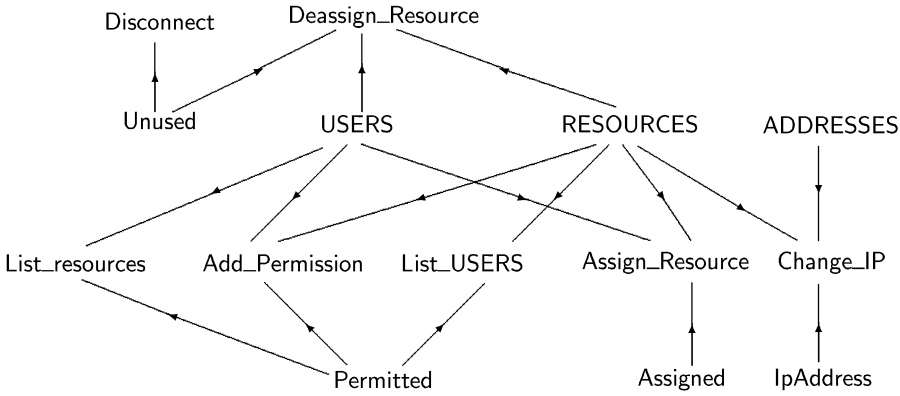

Fig. 11. *The complete context representation of the AccessControl specification*

The concept dependence context showed by the figure 11 has the following properties:

- $\max(\mathcal{G}) = \{RESOURCES, USERS, Unused, Permitted\}$ leading to select *RESOURCES*, *USERS*, and *Unused* as classes for the new diagram.
- $\mathrm{Op}(\{ADDRESSES\}) \subseteq \mathrm{Op}(\{RESOURCES\})$ which shows the non pertinency of the class *ADDRESSES* and turns it into a type of the class *RESOURCES*;
- $\mathrm{Op}(\{IpAddress\}) \subseteq \mathrm{Op}(\{RESOURCES\})$ which shows the non pertinency of the relation *IpAddress* as an association and turns it into an attribute of the class *RESOURCES*.

Finally, the resulting class diagram is given in Fig. 12

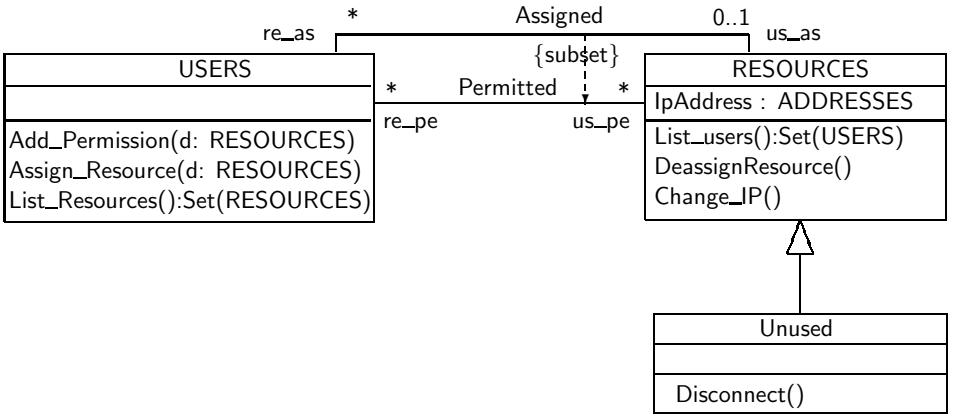

Fig. 12. *The Final pertinent class diagram for the AccessControl Machine*

5 Conclusion and perspectives

Although formal methods provide excellent techniques for the precise description of systems, understanding these descriptions is often restricted to experts. This paper has presented a technique that helps build a graphical representation of the static aspects of B specifications. These diagrams are expected to be more intuitive and readable than the original formal specification.

This paper has proposed a two-step approach to the construction of a class diagram from a B specification. The first step applies systematic transformation rules to the B specification and produces a preliminary class diagram whose goal is to identify candidate concepts. The second step relates these concepts to the B operations and keeps the most pertinent ones. This second step is original with respect to existing approaches such as [7,21]. We have led several case studies with the three approaches, including the one presented in this paper. In each case, we felt that our approach gave at least as good results as the other ones.

Still, our approach suffers several limitations:

- The major limit of the resulting diagram is that it gives a less complete information than that which could be expressed in a formal specification. Other views, such as the dynamic views constructed by [2,17], are needed to provide a complete graphical documentation of the B specification.
- The starting point of our approach is a single B machine. Further work is needed to address specifications which feature several machines, or which involve refinements.
- We only exploit a subset of the primitives of UML class diagrams. Constructs such as associative classes, aggregation and composition could probably enrich our approach.

- Further experimentations are needed to help us understand how our approach scales up.

In the coming months, we intend to address several of these limitations. We also plan to experiment with alternate reverse-engineering methods which address concept formation.

The successful integration of formal methods with existing graphical notations is important for the long-term success of formal methods. Industry will not abandon its current practices, but it is willing to augment and enhance them.

We believe that approaches like ours are a modest contribution to bridge the gap between current industry practices and the proposals of the formal method community.

References

[1] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. METEOR: A successful application of B in a large project. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99: World Congress on Formal Methods*, number 1709 in Lecture Notes in Computer Science, pages 369–387. Springer, 1999.

[2] D. Bert and F. Cave. Construction of Finite Labelled Transition Systems from B Abstract Systems. In *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[3] J.C. Bicarregui. Formal methods into practice: case studies in the application of the B method. *IEE Proceedings on Software Engineering*, 144(2):119–133, April 1997.

[4] L. Casset. Development of an embedded verifier for java card byte code using formal methods. In *FME'02, Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[5] J. Dick and J. Loubersac. Integrating structured and formal methods: A visual approach to VDM. In A. van Lamsweerde and A. Fugetta, editors, *Proceedings of European Software Engineering Conference (ESEC '91)*, volume 550 of *Lecture Notes in Computer Science. Springer-Verlag*, pages 37–59, October 1991.

[6] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Vers une intégration utile de notations semi-formelles et formelles : une expérience en UML et Z. *L'objet, numéro thématique Approches formelles à objets*, 6(1), 2000.

[7] Houda Fekih, Leila Jemni, and Stephan Merz. Transformation des spécifications B en des diagrammes UML. In *AFADL : Approches Formelles dans l'Assistance au Développement de Logiciels*, 2004.

[8] A. Hammad, B. Tatibouet, J.C. Voisinet, and Wu Weiping. From a B specification to UML statechart diagrams. In *4th International Conference on Formal Engineering Methods (ICFEM'2002), LNCS 2495*, pages 511–522, China, 2002.

[9] D. Harel. STATECHARTS: a visual formalism for complex systems. *Science of Computer Programming*, 8(3), 1987.

[10] A. Idani. Documentation graphique des projets B. Rapport de DEA, Univ. Joseph Fourier, Grenoble, France, 2003.

[11] R. Laleau and A. Mammar. An overview of a method and its support tool for generating B specifications from UML notations. In *International Conference on Automated Software Engineering (ASE2000)*. IEEE CS Press, 2000.

[12] Régine Laleau and Fiona Polack. Coming and going from UML to B: A proposal to support traceability in rigorous IS development. In *ZB'2002 – Formal Specification and Development in Z and B*, pages 517–534, 2002.

[13] K. Lano. *Formal object-oriented development*. Springer, 1995.

[14] K. Lano and S. Goldsack. Intregrated Formal and Object-Oriented Methods: The VDM++ Approach. In A.Bryant and L.Semmens, editors, *Method Integration Workshop*, Electronic Workshop in Computing, Leeds, Mars 1996. Springer-Verlag.

[15] K. Lano, H. Houghton, and P. Wheeler. Integrating Formal and Structured Methods in Object-Oriented System Development. In *Formal Methods and Object technology*, chapter 7. Springer, 1996.

[16] Hung Ledang and Jeanine Souquières. Contributions for modelling UML state-charts in B. In *Integrated Formal Methods, Third International Conference, IFM 2002*, volume 2335 of *Lecture Notes in Computer Science*, pages 109–127, 2002.

[17] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[18] Emil Sekerinski. Graphical design of reactive systems. In *B'98: The 2nd International* B *Conference, Recent Advances in the Development and Use of the* B *Method*, volume 1393 of *Lecture Notes in Computer Science. Springer-Verlag*, 1998.

[19] C. Snook and M. Butler. Using a graphical design tool for formal specification. In *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group*, 2001.

[20] J.M. Spivey. *The Z notation - A Reference Manual (2nd Ed.)*. Prentice Hall, 1992.

[21] B. Tatibouet, A. Hammad, and J.C. Voisinet. From an abstract B specification to UML class diagrams. In *2nd IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'2002)*, Marrakech, Morocco, December 2002.

[22] B. Tatibouet and J.C. Voisinet. Generating statecharts from B specifications. In *16th International Conference on Software and Systems Engineering and their applications (ICSSEA'2003)*, CNAM - Paris, France, December 2003.

[23] J. Warmer and A. Kleppe. *The Object Constraint Langage: Precise Modeling with UML*. Addison Wesley, Reading, Mass., 1999.