Modeling class operations in B: application to UML behavioral diagrams

Hung LEDANG and Jeanine SOUQUIÈRES

LORIA - Université Nancy 2 - UMR 7503
Campus scientifique, BP 239
54506 Vandœuvre-les-Nancy Cedex - France
E-mail: {ledang,souquier}@loria.fr

Abstract

An appropriate approach for translating UML to B formal specifications allows one to use UML and B jointly in an unified, practical and rigorous software development. We formally analyze UML specifications via their corresponding B formal specifications. This point is significant because B support tools like AtelierB are available. We can also use UML specifications as a tool for building B specifications, so the development of B specifications become easier.

In this paper, we address the problem of automatic derivation from UML behavioral diagrams into B specifications, which has been so far an open issue. A new approach for modeling class operations in B is presented. Each class operation is mapped into a B operation. A class operation and its involved data are mapped into the same B abstract machine (BAM). The class operation calling-called dependency is used to arrange derived B operations into BAMs. For each calling-called pair of class operations, the B operation of the called operation participates in the implementation of the B operation of the calling operation.

Keywords: *UML, class operation, B method, B abstract machine(BAM), B operation.*

1 Introduction

The Unified Modeling Language (UML)[14] has become a de-facto standard notation for describing analysis and design models of object-oriented software systems. The graphical description of models is easily accessible. Developers and their customers intuitively grasp the general structure of a model and thus have a good basis for discussing system requirements and their possible implementation. However, the fact that UML lacks a precise semantics is a serious drawback of UML-based techniques.

On the other hand, B[1] is a formal software development method that covers software process from the abstract specification to the executable implementation. A strong point of B (over other formal methods like Z and VDM) is support tools like AtelierB [16], B-Toolkit [2]. Most theoretical aspects of the method, such as the formulation of proof obligations, are done automatically by tools. Provers are also designed to run automatically and reference a large library of mathematical rules, provided with the system. All of these points make B be well adapted in large scale industrial projects [3]. However, as a formal method, B is still difficult to learn and use.

As pointed out in the literature [15, 5], an appropriate combination of object-oriented techniques and formal methods can give rise a practical and rigorous software development. For this objective, a promising approach is to derive B specifications from UML specifications [12, 13]. This allows one to rigorously verify UML specifications by analyzing derived B specifications, thanks to powerful support tools of B.

The perspectives to analyze UML specification via the derived B specification can be found in [8]. Afterwards, we only consider the problem of the automatic UML-to-B derivation. The dissertations of Meyer [12] and Nguyen [13] presented a set of rules for mapping UML static diagrams into B. Certain elements in UML behavioral diagrams like state and transition were also considered. So far, the problem of modeling UML behavioral diagrams in B has been an open issue since existing proposals to formalize in B class operations, events and use cases are not appropriate. For instance, they could not be applied to model in B class operations involving data from several classes.

In this paper, we first present an approach for modeling class operations in B. Then we show the way to translate

UML behavioral diagrams (interaction and activity) into B specifications. Like the work of Meyer and Nguyen, we model each class operation by a B operation. But our approach differs from theirs by proposing to group the class operation and its involved data in the same BAM. Thus, the problem of modeling in B the class operation effect is solved. Furthermore, to conserve the modularity of derived B specifications, we use the *class operation calling-called dependency*[1] to arrange derived B operations into different BAMs.

Section 2 introduces an example, which is used through the whole presentation. Section 3 recalls main achievements of the research in the UML-B derivation and approaches the problem of modeling in B class operations. Section 4 presents intuitively our ideas for modeling class operations in B. A procedure for automatically deriving B specifications from UML specifications is presented in Section 5. Discussions in Section 6 conclude our presentation.

2 Case study : the pump component

In an extended version of this paper [9] we presented an UML specification of the pump component. This specification is extracted from a case study of a system controlling petrol dispensing, customer payment handling and petrol tank level monitoring as described in chapter 6 of [6]. We have only developed the class and collaboration diagrams. The class diagram provides the structure of the component, while collaboration diagrams describe the global behavior of the component. For reasons of space, we introduce here only the class diagram in this UML specification as described in Figure 1.

The class operation calling-called dependency in Figure 2 is automatically derived from collaboration diagrams in [9]; the name of each class operation is preceded by the class name and "::" in order to clearly distinguish the operations with the same name from different classes; for reasons of space, we omit operation arguments. Notice also that the operations written in bold italic letters are derived from the aggregation amongst classes in the class diagram.

3 UML-B derivation

3.1 The B Method

B [1] is a formal software development method that covers the software process from specification to implementation. The B notation is based on set theory, the language of generalized substitutions and first order logic. Specifications are composed of BAMs similar to modules or classes;

[1] A calling-called pair relates a class operation - the *calling operation* - to one of its realization class operations - the *called operation.*

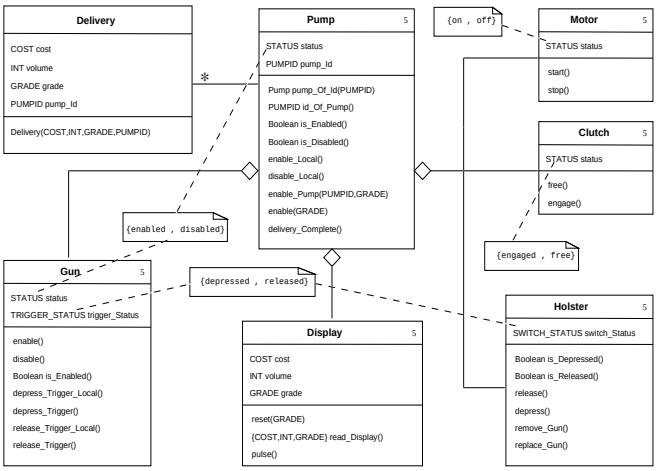

Figure 1. Class diagram of the pump component

Calling operations	Called operations
Pump::enable_Pump	Pump::pump_Of_Id Pump::enable
Pump::enable	Pump::is_Disabled Pump::enable_Local Display::reset Clutch::free Motor::start ***Pump::displayOfPump*** ***Pump::clutchOfPump*** ***Pump::motorOfPump***
Holster::remove_Gun	Holster::release Gun::enable ***Holster::pumpOfHolster*** ***Pump::gunOfPump***
Gun::depress_Trigger	Gun::depress_Trigger_Local Pump::is_Enabled Clutch::engage Gun::is_Enabled ***Gun::pumpOfGun*** ***Pump::clutchOfPump***
Gun::release_Trigger	Gun::release_Trigger_Local Clutch::free Gun::is_Enabled Pump::is_Enabled ***Gun::pumpOfGun*** ***Pump::clutchOfPump***
Holster::replace_Gun	Holster::depress Gun::disable Pump::delivery_Complete Pump::is_Enabled ***Holster::pumpOfHolster*** ***Pump::gunOfPump***
Pump::delivery_Complete	Pump::disable_Local Motor::stop Display::read_Display Delivery::Delivery Pump::idOfPump ***Pump::displayOfPump*** ***Pump::motorOfPump***
Display::pulse	

Figure 2. The calling-called dependency of class operations in Figure 1

they consist of variables, invariance properties relating to those variables and operations. The state of the system, i.e. the set of variable values, is only modifiable by operations. BAMs can be composed in various ways. Thus, large systems can be specified in a modular way, possibly reusing parts of other specifications. B refinement can be seen as an implementation technique but also as a specification technique to progressively augment a specification with more details until an implementation that can be translated into a programming language like ADA, C or C++. At every stage of the specification, proof obligations ensure that operations preserve the system invariant. A set of proof obligations that is sufficient for correctness must be discharged when a refinement is postulated between two B components.

3.2 State of the art

In [12, 13], Meyer and Nguyen proposed a set of precise rules for mapping UML class diagrams into B. Each `Class` class is formally derived by a *Class* BAM. *Class* declares a *CLASS* B deferred set that models the instance space of `Class`. The set of the effective instances of `Class` is modeled by a *class* B variable constrained to be a subset of *CLASS*. For each `attr` attribute, a *class_attr* B variable[2] is created and defined in the INVARIANT clause as a binary relation between *class* and the *Type_attr* B set modeling the `Type_attr` type of `attr`. This binary relation may be refined in a more sophisticated relation, such as function, bijection etc, according to the additional features of `attr`. Figure 3 shows a BAM and its data which are derived from the `Holster` class presented in Figure 1.

MACHINE *Holster*
SETS
HOLSTER;
HOLSTER_SWITCH_STATUS = { *holster_depressed,holster_released* }
VARIABLES
holster,
holster_switch_Status
INVARIANT
holster $\subseteq$ *HOLSTER* $\wedge$
holster_switch_Status $\in$ *holster* $\rightarrow$ *HOLSTER_SWITCH_STATUS*
...
END

Figure 3. Modeling in B the `Holster` class data

An `ass` association between `Class1` and `Class2` classes is identified by couples of instances. It is naturally expressed as a *ass* B variable of type of the binary relation (maybe a more sophisticated relation according to the multiplicity and other constraints of association) between *class1* and *class2* B variables. If `ass` is a non-fixed association[3] then `ass` gives rise to a new BAM, otherwise *ass* B variable is attached to one of *Class1* or *Class2* BAMs. The aggregation and composition has also been modeled in B according to their semantics described in [14]. As an example, the aggregation between `Pump` and `Holster` classes in Figure 1 is expressed as the *holsterPump* B variable (Figure 4), which is a bijection from the *pump* B variable into the *holster* B variable.

For the inheritance relationship, the current work of Meyer and Nguyen deal appropriately with domain classes but not with design and implementation classes. The B variable of a subclass in a specialization hierarchy is a subset of the B variable of its superclass. The BAM of a subclass "USES" the BAM of its superclass. Examples for the formalization in B of inheritance can be found in chapter 6 of Meyer's dissertation [12].

The important idea in the work of Meyer and Nguyen for modeling in B class diagrams is that one BAM is created for each class: attributes are modeled as B variables; class operations become B operations in the BAM. At first glance, this seems evident, however, at a closer inspection, the concept of class and the concept of BAM do not coincide with each other. A class operation can affect the data from different classes but a B operation affects only data declared in the same BAM. For this reason, only simple class operations like constructor, destructor or operations that set or query the value of each attributes (selector and mutator), which are local to classes, can be actually modeled. We could not model non basic class operations involving several classes. Consider the modeling in B of the `Holster` :: `remove_Gun` class operation, which modify `Holster` :: `switch_Status` and `Gun` :: `status` variables (cf. the schema operation of `Holster` :: `remove_Gun` in [6]). In the *Holster* BAM, it is impossible to access and modify the *gun_status* B variable from the *Gun* BAM. In the BAM, which includes *Holster* and *Gun*, it is not possible to explicitly express modifications of *holster_switch_Status* and *gun_status* B variables declared in the included BAMs. Moreover, since the existing proposals only used the BAM construct and the B inclusion mechanism, we cannot model the sequential calls of operations in collaboration or activity diagrams realizing non basic class operations. Consequently, the realization of non-basic class operations is also glossed over. Thus, with the current UML-B derivation schemes, we cannot translate interaction diagrams like sequence and collaboration to B.

4 Modeling class operations in B

4.1 Grouping data and operation

By grouping a class operation and its related data in the same BAM, the problem of modeling class operations in B becomes one of how B substitutions can be used to express the pre-/post condition of the operation. This is similar to model in B basic operations as described in the work of Meyer and Nguyen. Figure 4 shows the *MachineA* BAM which contains the *holster_remove_Gun* B operation corresponding to the `Holster` :: `remove_Gun` class operation; in the data declaration section (SETS, VARIABLES and INVARIANT clauses) of *MachineA* we notice the presence of data which are derived from different classes related to `Holster` :: `remove_Gun`; those are `Holster`, `Gun`, `Pump` and their associations.

We may create one BAM for the whole set of collaborating classes of a component's UML specification[4]; the BAM

[2] We use class name as the prefix for the B name of the elements inside a class in order to clearly distinguish the elements having the same name from different classes.

[3] The association between two classes whose instances are independently created/deleted in comparison with the instances of related classes.

[4] We consider here a component's UML specification consists of classes

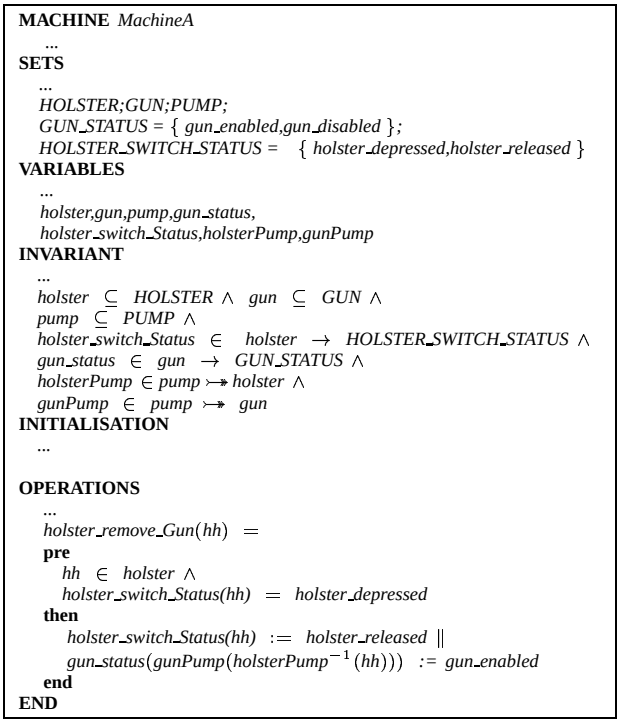

Figure 4. *Holster_remove_Gun* **B operation models** `Holster :: remove_Gun` **class operation**

data are derived from the whole class diagram and the operations are all class operations. However, grouping all the class operations in the same BAM prevents us from modeling the class operation calling-called dependency; for example, if `Holster :: release` and `Holster :: remove_Gun` are modeled in the same BAM then we are not able to model the fact that `Holster :: release` appears in the realization of `Holster :: remove_Gun`. This is because a B operation cannot call another one in the same BAM. In other words, we must create several BAMs for class operations in order to model the class operation calling-called dependency.

4.2 Modeling the class operation calling-called dependency

The intuitive idea is to separate a calling operation from its called operations; if `OpA` and `OpB` class operations form a calling-called pair, then `OpA` and `OpB` are modeled in two different BAMs that we call *MachineA* and *MachineB*. In the implementation of *MachineA* we import *MachineB* so we can call the *OpB* B operation in the implementation of the *OpA* B operation; in the case of neither `OpA` nor `OpB` calling the other, they are independent and we can model them either in the same BAM or in two BAMs; if `OpA` and `OpB` come from the same class, it is recommended to group them in the same BAM (this is the case for basic operations of a class).

In Figure 5 the *MachineA* BAM is implemented in the *MachineA_imp* B implementation that imports the *MachineB* BAM. In *MachineB* we model `Gun :: enable`, `Holster :: release`, `Holster :: pumpOfHolster` and `Pump :: gunOfPump` class operations that are called operations of `Holster :: remove_Gun` (Figure 2). As we can see, data in *MachineB* are identical to *MachineA* data since they are all derived from the same classes involved by `Holster :: remove_Gun`. This point is explicitly asserted in the INVARIANT clause of *MachineA_imp*. For this purpose, and by the nature of the B language, *MachineB* is renamed in the IMPORTS clause of *MachineA_imp* so that we can distinguish two set of data in *MachineA* and in *MachineB*. Several remarks should be made:

- the similar idea has been used in our previous work for modeling use cases [7]. We can also extend this approach to model events in state-chart diagrams [10];
- we have recently discovered that the B refinement construct and B inclusion mechanism are also appropriated to model the class operation calling-called dependency. However, in this paper we only speak of using B implementation/import dual;
- our approach for modeling in B the class operation calling-called dependency is only appropriate if there is no circular calling-called dependency amongst class operations. Consider three class operations $\texttt{Op}_1$, $\texttt{Op}_2$ and $\texttt{Op}_3$. Assume that: $\texttt{Op}_1$ calls $\texttt{Op}_2$; $\texttt{Op}_2$ calls $\texttt{Op}_3$ and $\texttt{Op}_3$ calls $\texttt{Op}_1$. So the BAM for $\texttt{Op}_1$ is implemented by importing the BAM of $\texttt{Op}_2$ which in turn is implemented by importing the BAM of $\texttt{Op}_3$. Because $\texttt{Op}_3$ calls $\texttt{Op}_1$, the BAM of $\texttt{Op}_3$ is implemented by importing the BAM of $\texttt{Op}_1$. This situation is impossible in B [1];
- by using the B implementation/import dual to model the class operation calling-called dependency, we cannot implement the concurrency inside class operations. This is due to restrictions of B with respect to the implementation construct. Indeed, in a B implementation operation we cannot express two operation calls concurrently. We have thought about using B refinement/includes to deal with this problem, however this discussion is not the purpose of this paper.

Apart from circular class operation calling-called dependency and without the concurrency inside class operations, we have proposed two procedures to arrange class operations into layers from which BAMs are created (cf. Section

whose object collaborate with each other.

```
IMPLEMENTATION MachineA_imp
REFINES MachineA
IMPORTS im.MachineB
INVARIANT
  holster = im.holster ∧
  gun = im.gun ∧
  pump = im.pump ∧
  ...
OPERATIONS
    ...
    holster_remove_Gun(hh) =
    var pp,gg in
        pp ← im.pumpOfHolster(hh);
        gg ← im.gunOfPump(pp);
        im.holster_release(hh);
        im.gun_enable(gg)
    end
END


MACHINE MachineB

  /* the data in MachineB are identical to the data in MachineA */

OPERATIONS
  ...
  holster_release(hh) = pre hh ∈ holster ∧
    holster_switch_Status(hh) = holster_depressed
  then
    holster_switch_Status(hh) := holster_released
  end;
  gun_enable(gg) = pre gg ∈ gun
  then
    gun_status(gg) := gun_enabled
  end;
  pp ← pumpOfHolster(hh) =
  pre hh ∈ holster then
    pp := holsterPump^-1(hh)
  end;
  gg ← gunOfPump(pp) =
  pre pp ∈ pump then
    gg := gunPump(pp)
  end
END
```

Figure 5. Modeling the class operation calling-called dependency

5): (i) the **division procedure** divides the class operations into layers such that operations in the same layer are independent of each other and they only depend on operations in lower layers; and (ii) the **duplication procedure** modifies the operation layers obtained from the division procedure such that operations in one layer, which differs from the bottom layer, have only called operations in the next lower layer.

4.3 Division procedure

1. **Intuitive idea**

   (a) **Creating the top layer:** all the operations which do not have any calling operations but have some called operations form the top layer.

   (b) **Creating the bottom layer:** all the operations which do not have any called operations form the bottom layer.

   (c) **Creating intermediate layer(s):** from the top layer, we find all operations which have only the calling operations in the top layer and also have some called operations; if there is no such operation (i.e we encounter the bottom layer) then we should stop, otherwise the obtained operations form the first intermediate layer.

   We repeat this step but this time we find the operations which have calling operations in the top layer or in the previous intermediate layers until we encounter the bottom layer.

2. **Application to the case study**

   It is easy to check that there is no circular class operation calling-called dependency in Figure 2. By applying the division procedure on this set of class operations, we obtain three operation layers: the top, the bottom and one intermediate layer as represented in Figure 6; in this Figure, each arrowed line comes from a calling operation to one of its called operations.

From operation layers in Figure 6, if we create one BAM for each layer then we have three BAMs: *SystemMachine* for operations in the top layer; *IntermediateMachine* for the intermediate layer and *BasicMachine* for the bottom layer. However, there is still a problem. Indeed, the operations modeled in *SystemMachine* depend at the same time on operations modeled inside *IntermediateMachine* and *BasicMachine*. Thus, both *IntermediateMachine* and *BasicMachine* are imported in the implementation of *SystemMachine*. In B, this is not allowed since *BasicMachine* is also imported in the implementation of *IntermediateMachine.*

To remedy such a situation which is often encountered in operation layers, certain operations should be duplicated in several layers as described in the duplication procedure.

4.4 Duplication procedure

1. **Intuitive idea**

   Let us introduce some conventions; given an `Op` operation, a `l` layer, we denote: `layer(Op)` the layer in which `Op` is found by applying the division procedure; `upper_than(l)` the set of upper layers of `l`; `next_upper(l)` the next upper layer of `l` (if `l` differs from the top layer).

   The goal of the duplication procedure is to duplicate several operations in several layers so that each `Op` operation is only called by operations from the `next_upper(layer(Op))` layer.

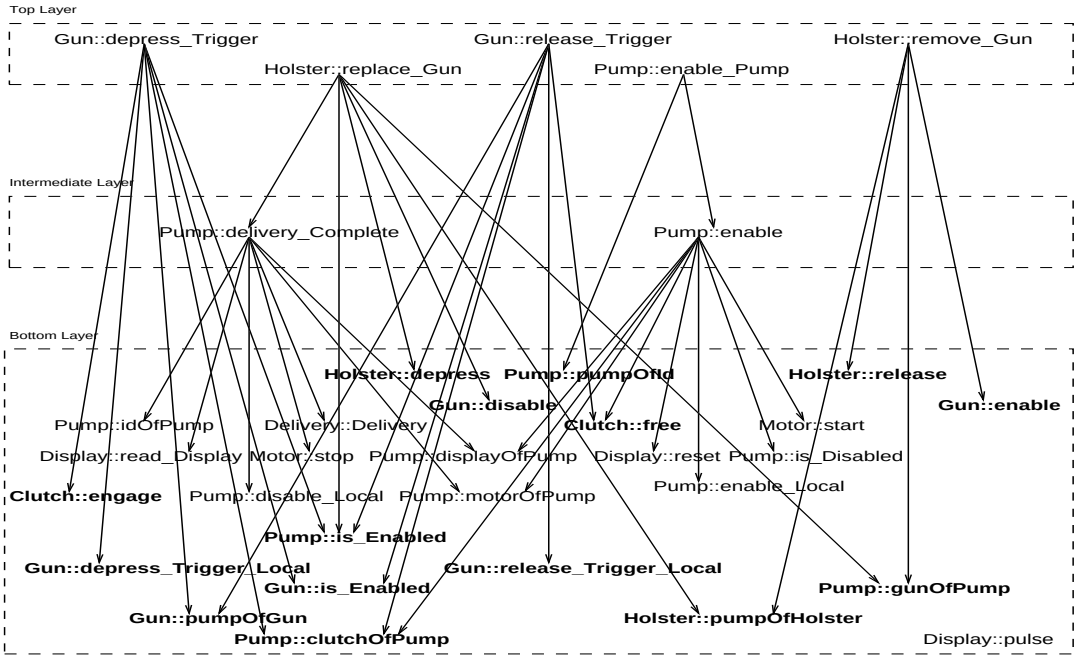

Figure 6. Pump operation layers obtained by the division procedure

(a) **Duplicating one operation in the next upper layer:** given an `Op` operation that is not in the top layer and is a called operation of some operations in upper layers. We add in the `next_upper(layer(Op))` layer an `Op_Dum` operation (the renaming is not necessary, however we use it to respect the previous work in [9]) which is identical to `Op`. We then replace all references from operations in `upper_than(layer(Op))` $-$ `{next_upper(layer(Op))}` layers to `Op` by the references to `Op_Dum`. We add also a reference from `Op_Dum` to `Op`. This special reference can be interpreted as the fact that `Op_Dum` and `Op` form a calling-called pair.

(b) **Duplicating one operation in several upper layers:** we repeat the above step for all applicable situations.

2. **Application to the case study**
In Figure 6, the operations written in bold letters are operations to be duplicated in the intermediate layer.

5 Developing B specifications from UML specifications

In this section we apply the division and the duplication procedures for developing the B specification of a component from its UML specification. As noticed earlier (Footnote 4 in Section 4), a component's UML specification consists of collaborating classes whose objects collaborate with each other in order to carry out the system operation [6] of the component.

5.1 The generic derivation procedure

1. **Dividing class operations into layers**

   (a) **Establishing the class operation calling-called dependency:** we browse all realization diagrams of class operations (usually the collaboration or activity diagrams [4]) and collect calling-called pairs of class operations.

   (b) **Checking non-circular class operation calling-called dependency:** we create an oriented graph, each node of which corresponds to a class operation. Each calling-called pair gives rise to an edge from the node of the calling operation to the node of the called operation. We use a graph algorithm to verify if the graph contains a cycle. The fact of having no cycle in the graph means that there is no circular class operation calling-called dependency; in that case we can continue in further steps, otherwise we must re-negotiate with the developer of the UML specification.

   (c) **Applying the division procedure.**

(d) **Applying the duplication procedure on the obtained class operation layers:** the class operation layers obtained in the previous step is updated by the duplication procedure to ensure that each operation is only called by operations in the next upper layer.

(e) **Applying the duplication procedure with orphan system operations:** sometimes we encounter in the bottom layer (or even in an intermediate layer) some system operations. Since they are system operations, we must model them in the *SystemMachine* BAM (cf. subsequent steps). However, according to their layer, they must be modeled in the *BasicMachine* BAM or in an *IntermediateMachine* BAM (cf. subsequent steps). To solve this conflict we use the duplication procedure to duplicate orphan system operations in all upper layers of its current layer.

2. **Creating *BasicMachine*:** we can create firstly BAMs for classes and non-fixed associations by using rules of Meyer and Nguyen. B operations in those BAMs are found in the bottom layer of class operation layers. The *BasicMachine* BAM is then created by including all BAMs for class and association. We also promote all operations of included machines.

3. **Creating *SystemMachine* and *IntermediateMachine(i)*:** operations in *SystemMachine* correspond to system operations of the component. In the class operation layers, those are operations on the top layer. We derive the data of *SystemMachine* from the whole class diagram of the component.

   For the intermediate layer number `i` (from the top layer) we create a BAM called *IntermediateMachine(i).* By definition, the data of *IntermediateMachine(i)* are data derived from the whole class diagram of the component. In the created BAM we model the operations from the associated layer.

4. **Implementing *SystemMachine* and *IntermediateMachine(i)*:** as stated earlier, *SystemMachine* and *IntermediateMachine(i)* (if any) are implemented by the BAM in the next lower layer. The implemented BAM and the imported BAM have identical data (because data in both BAM are all derived from the same class diagram). Hence, as noticed in section 4.2 the imported BAM is renamed (Figure 5) so that we have two distinct sets of data and one (of the imported BAM) implements (identically) the other (of the implemented BAM). The gluing invariant in implementation is used to assert the identity of two sets of data (Figure 5).

   Given `Op`, `Op_Dum` a duplicated-duplicating operation pair. The *Op_Dum* B operation is identical to *Op* B operation (duplication) in the created BAMs. But in the implementation of *Op_Dum* it is sufficient to call *Op*.

5.2 Application to the case study

In the complete UML specification given in [9], there are only collaboration diagrams acting as realization diagrams. The class operation calling-called dependency (Figure 2) is therefore derived from those collaboration diagrams.

The `Display` :: `pulse` operation (Figure 6) is the unique orphan system operation in our example. By applying the duplication procedure twice for this operation in the intermediate and the top layers we create `Display` :: `pulse_Dum` in the intermediate layer and `Display` :: `pulse_Dum_Dum` in the top layer.

Presently, we create three BAMs corresponding to three operation layers. *SytemMachine* for the top layer; *IntermediateMachine* for the intermediate layer and *BasicMachine* for the bottom layer. The *SystemMachine_imp* B implementation of *SystemMachine* imports *IntermediateMachine*(renamed); the *IntermediateMachine_imp* B implementation of *IntermediateMachine* imports *BasicMachine* (renamed).

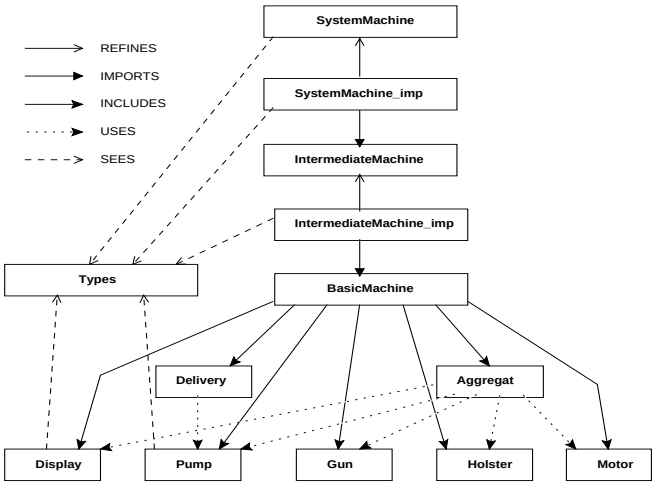

Figure 7. Architecture of the B specification for the pump component

Figure 7 represents the architecture of the B specification derived from class and collaboration diagrams of the pump component. For reasons of space, the code of the B specification, which is given in the extended version, is omitted in

this paper.

BasicMachine, by definition, includes BAMs derived from classes and associations. In our example, we create only one *Aggregat* BAM for the aggregation amongst the `Pump` class and its component classes. The association between `Pump` and `Delivery` is translated by the USES link from *Delivery* to *Pump* according to the rules given by Meyer and Nguyen.

As noticed in [9], `COST` and `GRADE` data types are defined in other components but they are referenced in `Delivery` and `Display` classes. Those data types are modeled in a special BAM called *Types* that is seen (the "SEES" link) by *Delivery*, *Display*, *SystemMachine* and *IntermediateMachine* BAMs where are modeled the data of `COST` and `GRADE` types. Furthermore, by definition *SystemMachine_imp* and *IntermediateMachine_imp* also "SEES" *Types*.

6 Conclusion

In this paper, we present an approach for modeling class operations in B. Our approach overcomes shortcomings of the existing approaches by taking into account: (i) the calling-called dependency amongst class operations and (ii) the binding between an operation and its concerned data. We also showed a way to translate UML realization and class diagrams into B specifications. Three procedures proposed in this paper are implementable to generate the architecture of B specifications from UML specifications. The data, the skeleton of B operations in the B specification are also automatically derived. In order to complete B specifications, we must fill up the body of B operations.

Automation of the generation of B operations is our further study objectives. For this purpose, we propose to attach to each class operation an OCL-based pre/-post specification. Hence, the abstract content of B operations can be derived by using OCL-B rules of Marcano and Lévy [11]. The implementation content of B operations for non-basic class operations is derived from realization diagrams of the considered operation. The precise rules will be envisaged in a later stage. In addition, the support tool for automatically translating class diagrams into B specifications developed by Meyer [12] will be extended to take into account UML behavioral diagrams.

References

[1] J.R. Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, 1996. ISBN 0-521-49619-5.

[2] B-Core(UK) Ltd, Oxford (UK). *B-Toolkit User's Manual*, 1996. Release 3.2.

[3] P. Behm, P. Desforges, and J.-M. Meynadier. MÉTÉOR: An Industrial Success in Formal Development, April 1998. An invited talk at the 2nd Int. B conference, LNCS 1939.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998. ISBN 0-201-57168-4.

[5] J.M. Bruel. Integrating Formal and Informal Specification Techniques. Why? How? In *the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, pages 50–57, Boca Raton, Florida (USA), 1998. Available at http://www.univ-pau.fr/ bruel/publications.html.

[6] D. Coleman, P. Arnold, St. Bodoff, Ch. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development : The Fusion Method.* Prentice Hall, 1994.

[7] H. Ledang. Des cas d'utilisation à une spécification B. In *Journées AFADL'2001 : Approches Formelles dans l'Assistance au Développement de Logiciels*, Nancy (F), 11-13 juin, 2001. http://www.loria.fr/∼ledang/publications/afadl01.ps.gz.

[8] H. Ledang and J. Souquières. Formalizing UML Behavioral Diagrams with B. In *the Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*, Tampa Bay, Florida (USA), October 15, 2001. http://www.loria.fr/∼ledang/publications/oopsla01.ps.gz.

[9] H. Ledang and J. Souquières. Modeling class operations in B : a case study on the pump component. Technical Report A01-R-011, Laboratoire Lorrain de Recherche en Informatique et ses Applications, March 2001. http://www.loria.fr/∼ledang/publications/UML01.ps.gz.

[10] H. Ledang and J. Souquières. New Approach for Modeling State-Chart Diagrams in B. Technical Report A01-R-082, Laboratoire Lorrain de Recherche en Informatique et ses Applications, September 2001. http://www.loria.fr/∼ledang/publications/state-chart-modeling.ps.gz.

[11] R. Marcano and N. Lévy. Transformation d'annotations OCL en expressions B. In *Journées AFADL'2001 : Approches Formelles dans l'Assistance au Développement de Logiciels*, Nancy (F), 11-13 juin, 2001.

[12] E. Meyer. *Développements formels par objets: utilisation conjointe de B et d'UML*. PhD thesis, LORIA - Université Nancy 2, Nancy (F), mars 2001.

[13] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, Conservatoire National des Arts et Métiers - CEDRIC, Paris (F), décembre 1998.

[14] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. ISBN 0-201-30998-X.

[15] C. Snook and R. Harrison. Practitioners Views on the Use of Formal Methods: An Industrial Survey by Structured Interview. *Information and Software Technology*, 43:275–283, March 2001.

[16] STERIA - Technologies de l'Information, Aix-en-Provence (F). *Atelier B, Manuel Utilisateur*, 1998. Version 3.5.