Modeling class operations in B: application to UML behavioral diagrams

Hung Ledang, Jeanine Souquières

Modeling class operations in B: application to UML behavioral diagrams

Hung LEDANG and Jeanine SOUQUIERES

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 integrating UML-B allows us to map UML specifications into B specifications. Therefore, we can formally analyze an UML specification via the corresponding B formal specification. This point is significant because B support tools are available. We can also use UML specifications as a tool for building B specifications, so the development of B specifications become easier. Hence, an approach for a practical and rigorous software development, which is based on UML and B, from the requirements elicitation to the executable code, could be achieved.

In this paper, we address the problem of automatic derivation of UML behavioral diagrams into B specifications, which has been so far an open issue. For this purpose, we propose a new approach for modeling class operations in B. Each class operation is mapped into a B operation. A class operation and its concerned data are mapped into the same B abstract machine (BAM). The calling-called dependency between class operations is used to arrange B operations of class 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, layered division of class operations, B method, B abstract machine(BAM), B operation.*

1 Introduction

The Unified Modeling Language (UML)[15] 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 object-oriented techniques based on UML.

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 to use.

As cited many times in the literature [12], an appropriate combination of object-oriented techniques and formal methods can give a way that is applicable in the software industry. For this objective, we advocate integrating UML and B specification techniques. Our approach is to propose derivation schemes from UML concepts into B notations. This UML-B integration has following advantages: (i) the construction of UML specifications is formally controlled; (ii) the construction of B specifications becomes easier with the presence of UML specifications. From the informal description of requirements, we successively build the object models with different degrees of abstraction. These models cover from conceptual models through logical design models to the implementation models of the software. This also means that the developed models are successively refined. We verify the consistency of each object model by analyzing the derived B specification. We verify the conformance amongst object models by analyzing the refinement depen-

dency amongst them that is formally expressed in B.

At the present, only the UML-B derivation has been considered. The problem of analyzing the derived B specification remains at a later stage. The works in [7, 12, 13, 14] presented a set of rules for mapping UML static diagrams into B. Certain elements in UML behavioral diagrams like state and transition have been partially treated. So far, the problem of modeling UML behavioral diagrams in B has been an open issue. In [6], Laleau and Mammar have presented a support tool for generating B specifications from UML diagrams of data intensive applications. Although they considered UML collaboration diagrams, nothing new is added with respect to Nguyen's work [14]. The main reason is that the existing works coincide the UML class with the BAM concept. But in fact, they do not coincide each with 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 basic class operations, which are local to classes, can be modeled. We cannot model class operations concerning several classes.

In this paper, we first present an approach for modeling class operations in B. Then we show how to apply this approach to translate UML behavioral diagrams into B specifications. Like the previous work [12, 14], we model each class operation as a B operation. But our approach differs from them by proposing to group the class operation and its concerned data in the same BAM. This combination allows us to overcome the problem of modeling class operations in the form of pre-/post condition in B operations. In addition, we also consider the calling-called dependency amongst class operations[1]. The B operation of the called operation participates in the implementation of the B operation which model the calling operation. That means that: (i) the BAM for the called operation is imported in the implementation of the BAM for the calling operation and (ii) we use B implementation operation to model the realization of class operations.

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

2 Case study : the pump component

In an extended version [9] of this paper we have 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 [5]. We have only developed the class and collaboration diagrams. The class diagram provides the structure of the component, while the collaboration diagrams describe the global behavior of the component and are used for establishing the calling-called dependency amongst class operations. For reasons of space, we introduce here only the class diagram in this UML specification as described in Figure 1.

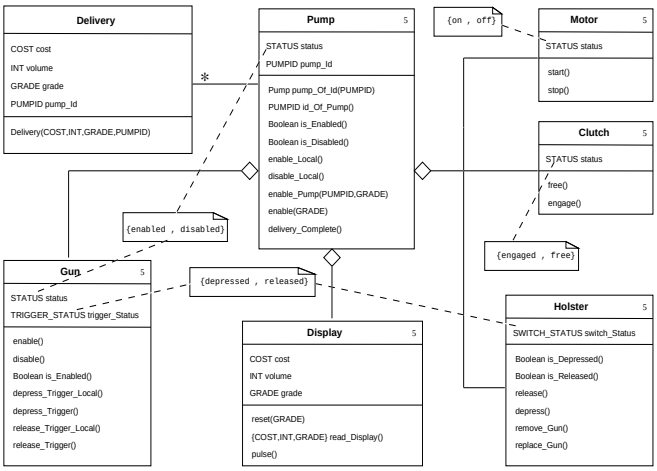

Figure 1. Class diagram of the pump component

The calling-called dependency amongst class operations as described 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 have omitted the operations' 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; they consist of a set 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.

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

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. Dependency between classes operations of the pump component

Thus, large systems can be specified in a modular way, possibly reusing parts of other specifications. Refinement of a B model allows developers to derive a correct implementation in a systematic way. Refinement can be seen as an implementation technique but also as a specification technique to progressively augment a specification with more details. 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, 14], Meyer and Nguyen have proposed a set of precise rules for mapping UML class diagrams into B. Each class `Class` is formally derived by a BAM *Class*. A BAM *Class* declares a B deferred set *CLASS*, which models the set of possible instances of the class `Class`. The set of the effective instances of the class `Class` is modeled by a B variable *class* constrained to be a subset of *CLASS*. For each attribute `attr`, a B variable *class_attr*[2] is created and defined in the INVARIANT clause as a binary relation between the B set *class* and a B set *Type_attr* modeling the type `Type_attr` associated with `attr`. This binary relation may be refined in the 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 class `Holster` presented in Figure 1.

An association `ass` between two classes `Class1` and `Class2` is identified by couples of instances. It is naturally expressed in B as a variable *ass* of the type of the binary relation (maybe a more sophisticated relation as noticed earlier) between B variables *class1* and *class2*. If `ass` is a non-fixed association[3] then `ass` gives rise to a BAM, otherwise the B variable *ass* is attached to one of the BAMs *Class1* or *Class2*. As an example, the aggregation between the classes `Pump` and `Holster` in Figure 1 is expressed as a B variable *holsterPump* (Figure 4), which is a bijection from the B variable *pump* of the class `Pump` into the B variable *holster* of the class `Holster`. For reasons of space, we have omitted here the rules concerning the inheritance.

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. A B representation for data of the class `Holster`

If the rules for modeling concepts of data aspects are formally defined and can be implemented in a piece of software, the rules for formalizing concepts of behavioral aspects must be intensively done. The main reason is that they have no appropriate solution for dealing with class operations, which is the core concept in the behavioral diagrams. In fact, with existing rules, we cannot, in general, model class operation concerning data of several classes. Consider the modeling of the operation `Holster::remove_Gun` which modify the variables `Holster::switch_Status` and `Gun::status` (as described in the schema operation of `Holster::remove_Gun` in [5]). In the BAM *Holster*, it is impossible to access and modify the B variable *gun_status* from the BAM *Gun*. In the BAM, which includes BAMs *Holster* and *Gun*, it is not possible to explicitly express modifications of the B variables *holster_switch_Status* and *gun_status* of the included BAMs. Moreover, because the existing proposals only used the BAM construct and B inclusion mechanism so that we cannot model the sequential calls of operations in collaboration or activity diagrams realizing non basic class operations. That means that the realization of non-basic class operations is also glossed over.

[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.

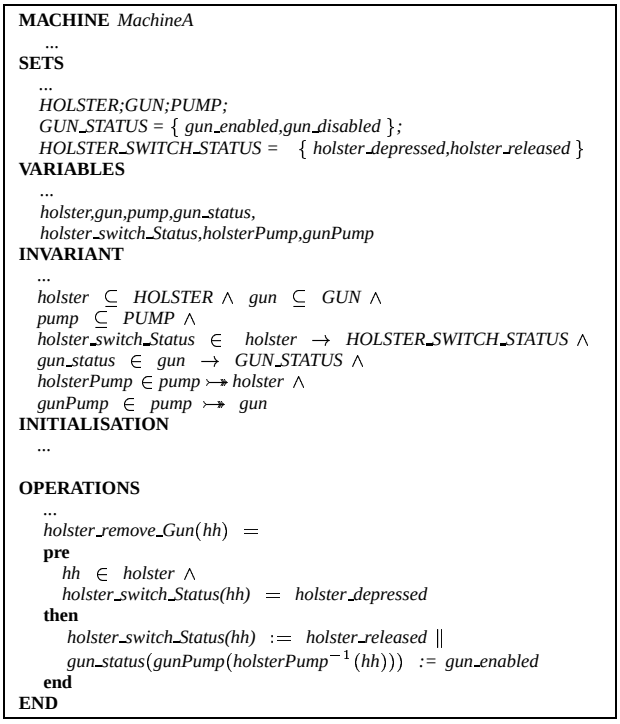

Figure 4. The B representation of the operation `Holster :: remove_Gun`

4 Modeling class operations in B

In [8], we have presented an approach for modeling use cases in B. Each use case is modeled as a B operation in a BAM whose data are derived from classes related to the use case. We believe this principle can be applied for modeling class operations. In addition, by using B implementation construct and B importation mechanism we can model the calling-called dependency amongst operations.

4.1 Grouping data and operation in the same BAM

By grouping a class operation and its related data in the same BAM, the problem of modeling class operations becomes one of how B substitutions can be used to express the pre-/post condition of the operation. This is similar to model basic operations as described in [12]. Figure 4 shows a BAM *MachineA* which contains the B operation *holster_remove_Gun* corresponding to the class operation `Holster :: remove_Gun`; in the data declaration section (clauses SETS, VARIABLES and INVARIANT) of *MachineA* we notice the presence of data which are derived from different classes related to the operation `Holster :: remove_Gun`; these are : `Holster`, `Gun`, `Pump` and their associations.

We may create a BAM for the whole set of collaborating classes of a component's UML specification[4]; the BAM 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 calling-called dependency amongst class operations; for example, if the operation `Holster :: release` is modeled in the same BAM as `Holster :: remove_Gun` 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 [1, 16]. In other words, we must create several BAMs for class operations in order to model the calling-called dependency amongst them. The following sections discuss how to arrange the class operations in BAMs.

4.2 Modeling the calling-called dependency amongst class operations

The intuitive idea is to separate a calling operation from its called operations; if two class operations `OpA` and `OpB` form a calling-called pair, then `OpA` and `OpB` are modeled in two different BAMs which we call *MachineA* and *MachineB*. In the implementation of *MachineA* we import *MachineB* so we can call B operation *OpB* in the implementation of the B operation *OpA*; 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 BAM *MachineA* of Figure 4 is implemented in the implementation *MachineA_imp* which imports the BAM *MachineB*. In *MachineB* we model operations `Gun :: enable`, `Holster :: release`, `Holster :: pumpOfHolster` and `Pump :: gunOfPump` being called operations of `Holster :: remove_Gun` (Figure 2). As we can see, the data in *MachineB* are identical to the data in *MachineA*; this is because they are all derived from the same class data concerned by `Holster :: remove_Gun`. This point is explicitly asserted in the INVARIANT clause of *MachineA_imp*. For this purpose, the BAM *MachineB* is renamed in the IMPORTS clause of *MachineA_imp* so that we can distinguish two set of data in *MachineA* and in *MachineB*[5]. Several remarks should be made:

- our approach for modeling the calling-called dependency relation amongst class operations is only appropriate if there is no cyclic calling-called dependency

[4] We consider here a component's UML specifi cation consists of classes whose object collaborate with each other.

[5] This is due to B.

```
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. An example of modeling the calling-called dependency amongst class operations

among class operations. Consider three class operations $\mathtt{Op_1}$, $\mathtt{Op_2}$ and $\mathtt{Op_3}$. Assume that: $\mathtt{Op_1}$ calls $\mathtt{Op_2}$; $\mathtt{Op_2}$ calls $\mathtt{Op_3}$ and $\mathtt{Op_3}$ calls $\mathtt{Op_1}$. So the BAM for $\mathtt{Op_1}$ is implemented by importing the BAM of $\mathtt{Op_2}$ which in turn is implemented by importing the BAM of $\mathtt{Op_3}$. Because $\mathtt{Op_3}$ calls $\mathtt{Op_1}$, the BAM of $\mathtt{Op_3}$ is implemented by importing the BAM of $\mathtt{Op_1}$. This situation is impossible in B [1, 16];

- there are, in general, two possibilities for modeling the calling-called dependency amongst class operations: (i) using B implementation construct and B importation mechanism and (ii) using B refinement construct and B inclusion mechanism. We prefer the first one due to the expressing capacity. In fact, in some cases we can use the B refinement/inclusion dual; this is the case, for example, when `OpA` modeled in *MachineA* which calls the `OpB` modeled in *MachineB* and calls no other operations modeled in *MachineB*; however if `OpA` also calls some other operations modeled in *MachineB* then the refinement/inclusion dual is not appropriate due to technical restrictions of the B inclusion mechanism [1, 16];

- note also that our approach for modeling class operations only works without concurrence inside class operations. This is due to restrictions of B with respect to the implementation construct. In fact in a B implementation operation we cannot express two operation call concurrently.

Apart from cyclic calling-called dependency and without the concurrence inside class operations, we have proposed two procedures which are used in section 5 for deriving B specifications from UML specifications: (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 **"dummy promoting" 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 cyclic calling-called dependency amongst class operations of the pump component as described 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 arrow 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: the BAM *SystemMachine* models operations in the top layer; the BAM *IntermediateMachine* for the intermediate layer and the BAM *BasicMachine* for the bottom layer. However, there is still a problem. Indeed, the operations modeled in BAM *SystemMachine* depend at one and the same time on operations modeled by *IntermediateMachine* and *BasicMachine*. Thus, both *IntermediateMachine* and *BasicMachine* are imported in the implementation of *SystemMachine*. This is not allowed according to [1, 16] because the BAM *BasicMachine* is also imported in the implementation of *IntermediateMachine*.

To remedy such a situation which is often encountered in operation layers, we use the "dummy-promoting" procedure as described in the following section.

4.4 "Dummy-promoting" procedure

1. **Intuitive idea**
   Let us introduce some conventions; given an operation `Op`, a layer `l`, 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 "dummy-promoting" procedure is to duplicate several operations in several layers so that each operation `Op` is only called by operations from layer `next_upper(layer(Op))`.

   (a) **Duplicating one operation in the next upper layer**
   Given an operation `Op` which has upper layers and is called by operations in layers $\texttt{upper\_than(layer(Op))} - \{\texttt{next\_upper(layer)}\}$. We add in layer `next_upper(layer(Op))` one operation `Op_Dum` which is identical to `Op`. We then replace all references from operations in layers $\texttt{upper\_than(layer(Op))} - \{\texttt{next\_upper(layer(Op))}\}$ 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. Figure 7 is obtained from Figure 6 by applying the "dummy-promoting" procedure.

5 Developing B specifications from UML specifications

In this section we apply the division and the "dummy-promoting" procedures for developing the B specification of a component from its UML specification. As noticed earlier (Note 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 [5] of the component.

Before presenting the procedure for building the B specification, some terminology must be introduced.

5.1 Terminology

Layered division of class operations. By applying the division procedure on class operations we obtain several layers of operations. This is called layered division of class operations. The layered division of class operations is often modified by the "dummy-promoting" procedure before being used for deriving BAMs.

Class machine and Association machine. A BAM derived from a class or a non-fixed association is called the class machine or association machine. In a class machine we model data and operations, which are found in the bottom layer of the layered division of class operations, of the related class.

Basic machine. We call basic machine *BasicMachine* the BAM for which data are derived from all classes and associations of the component and operations modeling the basic operations inside the bottom layer of the layered division of class operations. Thus, the basic machine is created by including all class and association machines. Operations of the basic machine are therefore promoted from included BAMs.

System machine. Operations in the system machine *SystemMachine* correspond to system operations of the component. In the layered division of class operations, these are operations on the top layer. There is only one system machine for each component. Hence the data in

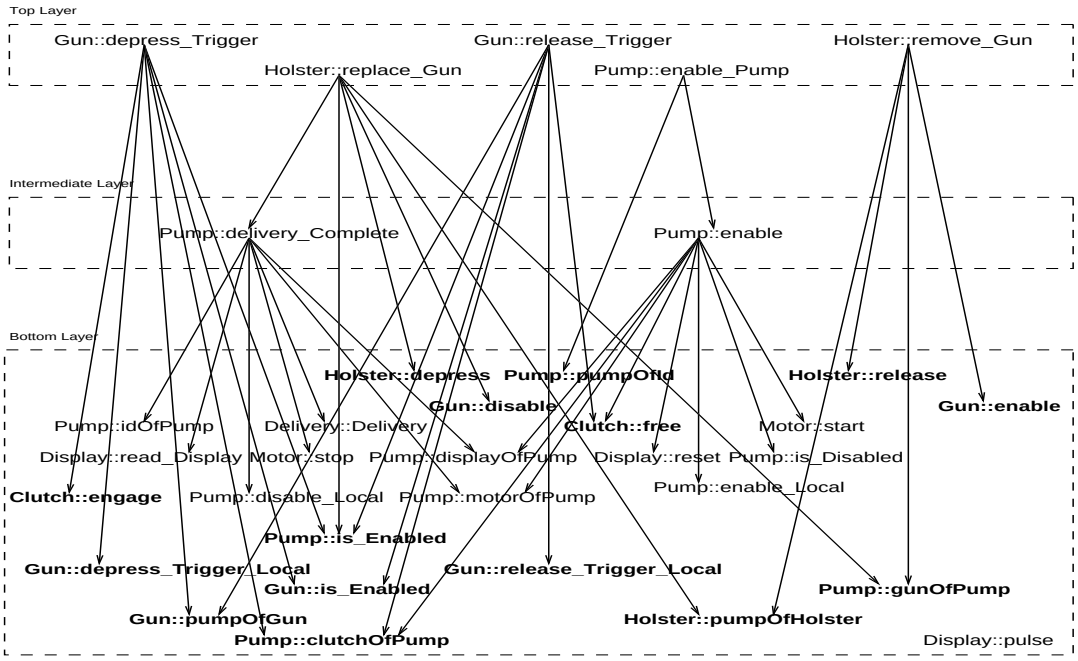

Figure 6. Pump operation layers obtained by the division procedure

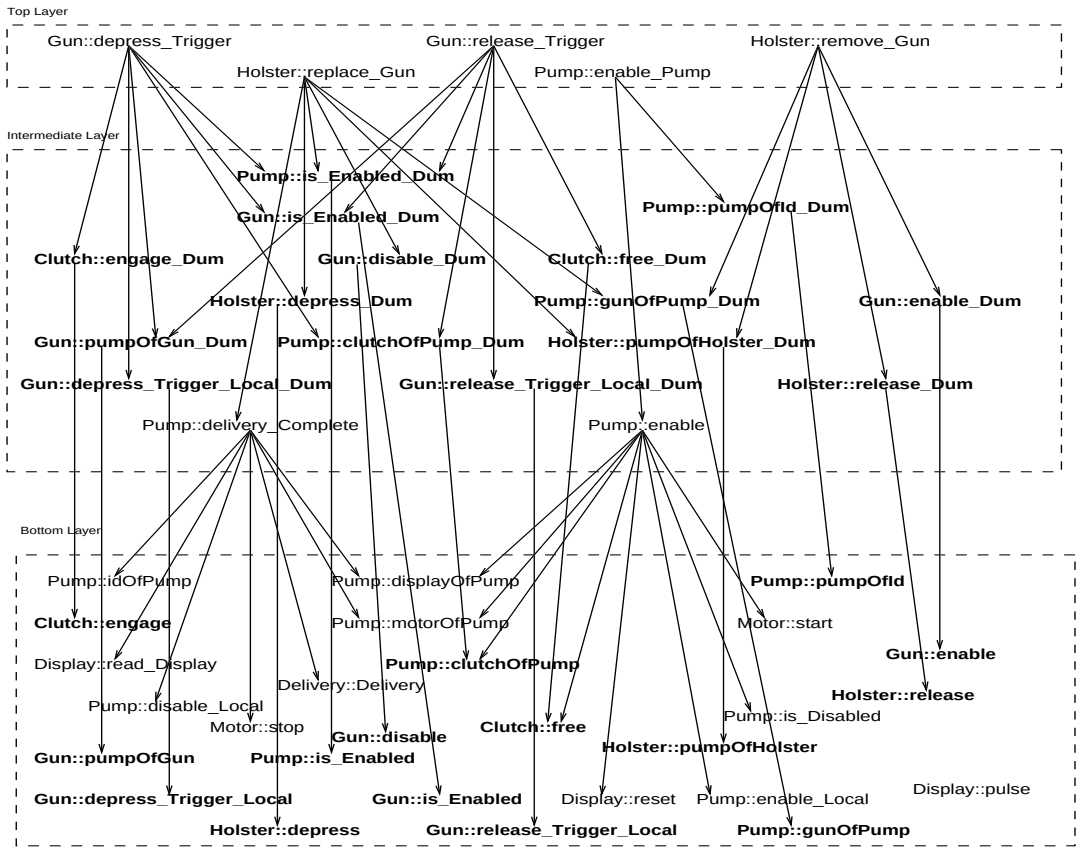

Figure 7. Pump operation layers after “dummy - promoting”

SystemMachine must be derived from the whole class diagram of the component.

Intermediate machine. If we have some intermediate layers in the layered division of class operations then we must create several BAMs *IntermediateMachine(i)* for these layers. Each intermediate BAM is for operations from one intermediate layer. *SystemMachine* is implemented by the intermediate BAM of the next lower layer from the top (if there is one) or by *BasicMachine* (if not). An intermediate BAM (if there is one) is implemented by the BAM of the next lower layer, which may be another intermediate BAM or *BasicMachine* if there is no intermediate layer lower than the intermediate layer of the BAM in question. Data in each intermediate BAM are also derived from the whole class diagram.

5.2 A generic procedure for developing the B specification

1. **Creating *BasicMachine***
   From the class diagram we can create all class and association machines by using rules from [12, 14]. *BasicMachine* is created by including all class machines and association machines. We also promote all operations of included machines. Let us note that all data, associated constraints and operations of *BasicMachine* are distributed in class and association machines.

2. **Dividing class operations into layers**

   (a) **Establishing the dependency amongst class operations**
   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-cyclic dependency amongst class operations**
   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 cyclic dependency amongst class operations; in that case we can continue in further steps, otherwise we must re-negotiate with the developer of the UML specification.

   (c) **Creating the preliminary layered division of class operations**
   We apply the division procedure to create the layered division of class operations.

   (d) **Applying the "dummy-promoting" procedure on the obtained layered division**
   The layered division of class operations obtained in the previous step is updated by the "dummy-promoting" procedure to ensure that each operation is only called by operations in the next upper layer.

   (e) **Applying the "dummy-promoting" procedure with orphan system operations**
   Sometimes we encounter in the bottom layer (or even in an intermediate layer) a system operation. Because they are system operations, we must model them in the system machine. For this purpose we use the "dummy-promoting" procedure to duplicate orphan system operations in all upper layers of its current layer in the layered division of class operations.

3. **Creating *SystemMachine***
   We derive the data of *SystemMachine* from the whole class diagrams of the component. We model all the operations in the top layer of the layered division in the BAM *SystemMachine*.

4. **Creating *IntermediateMachine(i)***
   For the intermediate layer number `i` (from the top layer) we create a BAM *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.

5. **Implementing *SystemMachine* and *IntermediateMachine(i)***
   As stated in sections 4.2 and 5.2, *SystemMachine* and *IntermediateMachine(i)* (if there are 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 a duplicated operation `Op` and its duplicating one `Op_Dum`. In BAMs, the B operation *Op_Dum* is identical to the B operation *Op* (duplication) but in the implementation of *Op_Dum* it is sufficient to call *Op*.

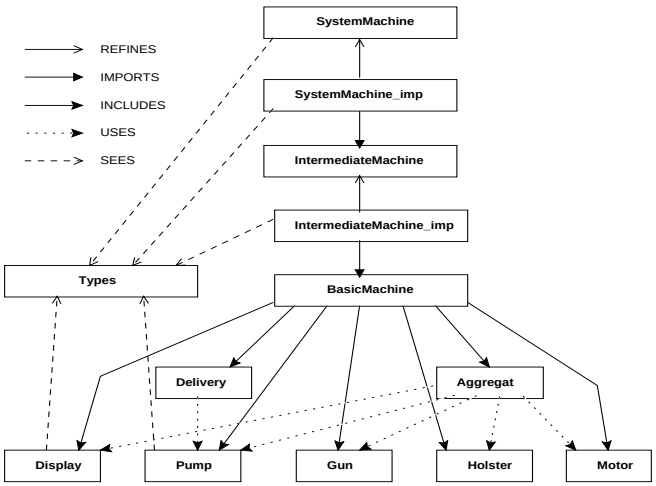

Figure 8. Architecture of the B specification for the pump component

5.3 Application to the case study

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

From the layered division of class operations obtained from the division and the "dummy-promoting" procedures (Figure 7), we notice that the operation `Display :: pulse` is an orphan system operation. We must apply the "dummy-promoting" procedure twice for this operation in the intermediate and the top layers. The result is to create `Display :: pulse_Dum` in the intermediate layer and `Display :: pulse_Dum_Dum` in the top layer.

At present, we create three BAMs corresponding to three operation layers. The BAM *SytemMachine* for the top layer; the BAM *IntermediateMachine* for the intermediate layer and the BAM *BasicMachine* for the bottom layer. The implementation *SystemMachine_imp* of *SystemMachine* imports *IntermediateMachine*; the implementation *IntermediateMachine_imp* of *IntermediateMachine* imports *BasicMachine.* As noticed in section 5.2 the BAMs *IntermediateMachine* and *BasicMachine* are renamed in the IMPORTS clauses.

Figure 8 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 given in [9] is omitted in this paper.

The BAM *BasicMachine* by definition includes the machines derived from classes and associations. In our example, we create only one association machine *Aggregat* for the aggregation amongst class `Pump` and its component classes. The association between `Pump` and `Delivery` is translated by the link USES from the BAM *Delivery* to the BAM *Pump* according to the rules given in [12, 14].

As noticed in the presentation of the case study in [9], the data types `COST` and `GRADE` are defined in other components but they are referenced in the classes `Delivery` and `Display`. These data types are modeled in a special BAM called *Types* which is seen (link "SEES") by the BAMs *Delivery*, *Display*, *SystemMachine* and *IntermediateMachine* because in these BAMs we model the data of the types `COST` and `GRADE`. In addition, the component *SystemMachine_imp* is the implementation of the BAM *SystemMachine* so by definition [1] it "SEES" also *Types*; the situation is similar for the component *IntermediateMachine_imp*.

5.4 Generating the content of B operations

It is easy to find that: corresponding to each non-basic class operation[6], there is a B abstract specification and a B implementation specification. The abstract content is in the BAM for the layer of the class operation and the implementation is in the corresponding implementation. The abstract content is made up by specification of the effect of class operation on the value of the manipulated objects. Whereas, in the implementation content, we model the realization of the considered class operation. Intuitively, each operation invocation in UML specification is translated to a B operation invocation in B specification.

At the present we can only automatically derive 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. For the purpose of a complete automation of transformation, 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 [10]. 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.

6 Conclusion

In this paper we have presented an approach to automatically integrate UML behavioral diagrams into B specifications. Our approach is based on three procedures:

division procedure, to divide class operations into layers according to the dependency amongst them;

"dummy-promoting" procedure, to modify the layered division obtained from the division procedure in order

[6] The class operation having a realization diagrams.

to ensure that operations in one layer differing from the bottom layer have only called operations in the next lower layer.

generic procedure, to translate UML specifications into B specifications. The generic procedure uses the division and "dummy-promoting" procedures to create the layered division of class operations. From these operation layers we automatically derive the architecture of the B specification. The data, the skeleton of B operations in the B specification are also automatically derived. It remains to fill up manually the body of B operations.

Our procedures can be implemented in a piece of software. The generic procedure provides a complete framework for deriving B specifications from UML structure and behavior diagrams. Hence, the conformance between two aspects (the structure and the behavior) of an UML specification can be formally verified by analyzing the corresponding B specification. This also means that we effectively have an appropriate and generalized solution for modeling in B the structure and the collaboration of design patterns which was mentioned in [12, 11] but only some typical patterns (the composite pattern, the client-server pattern) are treated.

For further work, a collaboration with Marcano and Lévy is envisaged to integrate their OCL-B translation rules [10] in our work (Section 5.4); a study to translate UML realization diagrams into B implementation operations is also envisaged. In addition, the support tool for automatically translating class diagrams into B specifications developped 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] D. Coleman, P. Arnold, St. Bodoff, Ch. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development : The Fusion Method*. Prentice Hall, 1994.

[6] R. Laleau and A. Mammar. An Overview of a Method and its support Tool for Generating B Specifications from UML Notations. In *The 15st IEEE Int. Conf. on Automated Software Engineering*, Grenoble (F), September 11-15, 2000.

[7] K. Lano. *The B Language and Method : A Guide to Practical Formal Development*. FACIT. Springer-Verlag, 1996. ISBN 3-540-76033-4.

[8] 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.

[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. Available at http://www.loria.fr/∼ledang/publications/UML01.ps.Z.

[10] 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.

[11] R. Marcano, E. Meyer, N. Lévy, and J. Souquières. Utilisation de patterns dans la construction de spcifications en UML et B. In *Journes AFADL'2000 : Approches Formelles dans l'Assistance au Développement de Logiciels*, janvier 2000.

[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] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *FM'99 : World Congress on Formal Methods in the Development of Computing Systems*, LNCS 1708, Toulouse (F), September 1999. Springer-Verlag.

[14] 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.

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

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