A Systematic Approach to Transform OMT Diagrams to a B Specification

Eric Meyer and Jeanine Souquières

LORIA - Université Nancy 2 - UMR 7503 ,
Campus scientifique, BP 239,
54506 Vandœuvre-les-Nancy Cedex - France,
Tel: +33 3.83.59.20.42 Fax: +33 3.83.41.30.79
{meyer,souquier}@loria.fr

Abstract. This paper presents a systematic transformation of semi-formal specifications expressed with OMT notations into formal specifications. The object model is first transformed into a specification composed of a set of B machines. Then each component of the dynamic model is transformed and integrated into the previous specification leading to a single specification. Transformations are presented as generic templates. When using these templates, the generated specification is automatically proved within the B prover relatively to the invariant preservation.

1 Introduction

Early phases of software development are crucial for the quality of software. The usefulness of formal specifications is now well accepted. But formal specifications are difficult to develop and to understand. More effort has been spent to develop new languages than to provide methodological guidance for using existing ones. But it does not suffice to hand a language description to specifiers and to expect them to be able to produce formal specifications. Moreover, formal specifications techniques are not well integrated with the analysis phase of software engineering.

Today, software engineers in industry mostly use semi-formal techniques like data-flow diagrams, entity-relationship diagrams, finite state machines or decision tables. Object oriented approaches and notations like OMT [15], Fusion [4] or UML [16] are now quite popular. These techniques have the advantage that they support the intuitive understanding of specification by graphical representations. They are useful for understanding problems and for the documentation allowing a better communication with customers. Sophisticated tool support is widely available. However, semi-formal techniques do not perform the transition from informal to formal texts. Therefore, the resulting specifications can be subject to misinterpretations. This makes them an insecure basis for the development contract.

Our approach aims at a better support of the early phases of the software development, thus contributing to a better quality of the specification. It votes

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 875–895, 1999.

to combine semi-formal diagrammatical descriptions with formal specifications, providing a bridge from the informal object oriented modelling concepts to the formal notation. It starts from the object and the dynamic models of the OMT method. Then these models are translated into a B specification [2] using predefined generic templates. We focus on producing formalisations that can directly support verification and validation activities, and for which mechanical support exists.

In this paper, we focus on the definition of the B generic templates issued from the object and dynamic OMT diagrams. Section 2 presents the OMT approach developing the object and the dynamic models. Section 3 is about the translation of these diagrams into a single B specification. Details are given on the definition of each component and on their integration into a single specification. Translations are defined as B generic predefined templates. In section 4, we discuss about the proof of these templates using the B tool. Section 5 presents the state of the art and a summary of our contributions concludes the paper.

2 An Overview of Part of the OMT Notations

The goal of OMT is to model systems using object-oriented concepts. The underlying method [15] integrates analysis, conception and coding activities, using three kind of models: the object, the dynamic and the functional ones.

The object model is the core of these model, it defines the static structure of the information manipulated by the application in terms of classes and the relationships among them. The dynamic model describes the evolution of the objects. It is defined by state diagrams. Automatons with a finite number of states are associated to all or a part of the classes. They present the different states of an object, transitions between these states and events that can be sent or received. The functional model defines the transformation of data in the system. Classically, it is issued from the data flow diagrams in the Yourdon and Ward [21] approach.

In this section, we summarise the different components of the object and dynamic models.

2.1 Object Model

Object model (see Fig. 1) provides a graphical notation for modelling objects, classes and their relationships to one another. They are easy to understand. The object model comes in two varieties of static diagrams: class diagrams showing generic descriptions of possible systems and object diagrams describing particular instantiations of systems. In most case, a system is built with several class diagrams plus occasional object diagrams illustrating complicated data structures.

Classes. A generic class diagram is presented in Fig. 1. Classes are drawn as boxes. A class ($Class_i$) describes a graph of objects with similar properties given

by a list of typed attributes ($AttrList_i$) and common behaviour expressed by a list of operations ($OperList_i$). A parameterized class is specified by including a list of formal typed parameters: $Class_i < Param_i : Type_{par} >$. Type and initial values for attribute can be specified as $Attr_i : Type_{att} = Val_{att}$. Input and output parameters of an operation can optionally be defined: $Op_i(arg : Type_{arg}) : Type_{op}$. The attribute and operation sections of the class box can be omitted to reduce detail in the overview. A class can have a multiplicity indicator ($mult_j$ for $class_j$), drawn as a small expression in the upper right corner; it indicates how many instances of the class can exist at a time.

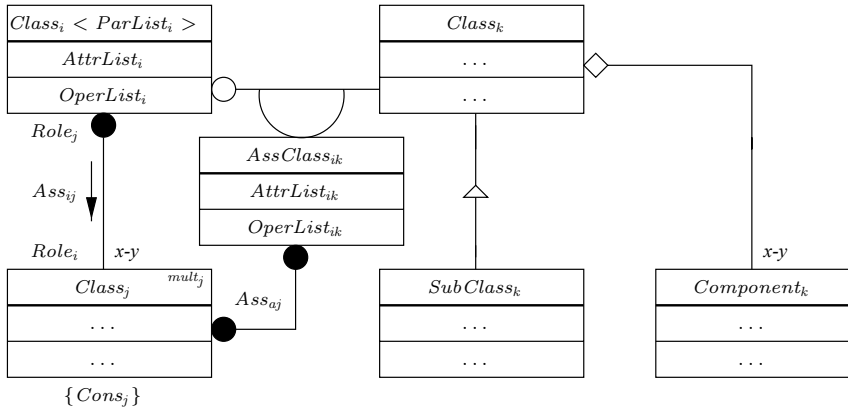


Fig. 1. A generic object model

Relations. Classes can be related to each other establishing structural relationships between objects of different classes. An association (Ass_{ij} in Fig. 1) relates two classes, either more by a line. It can have a name with an arrow showing which way the name is read. Each end of an association is a role ($Role_i, Role_j$). Each role can be named, describing how its class is considered by the other class. Each role indicates the cardinality of its class that restricts the number of class instances that can be associated to one instance of the other. Cardinality can be 1 (no marker), 0-1 (drawn by $\circ$), 0 or more (drawn by $\bullet$), or some other integer range value (indicated by an expression as $x - y$). An association can be considered as a class itself ($AssClass_{ik}$) with its own attributes, operations and associations. In addition to associations, inheritance relationship between a superclass ($Class_k$) and its subclass ($SubClass_k$) can be represented as a triangle with a line from the apex to the superclass and one line from the baseline to each subclass.

Aggregation. Aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the components of something are associated to an object representing the entire assembly. Aggregation is a tightly coupled form of

association with some extra semantics. Aggregations are drawn like associations, except a small diamond indicating the assembly end of the relationship. Aggregation is shown in Fig. 1 : $Class_k$ is defined by the aggregation $Component_k$.

Constraints. It is useful in some cases to restrict the values that objects or links can assume. A constraint is a restriction on values expressed as a predicate attached to a class or an association. It is enclosed in braces and placed near the elements it constrains (see $Cons_j$ for the $Class_j$ in Fig. 1).

2.2 Dynamic Model

The dynamic model (see Fig. 2) based on state diagrams is a complement to the object model. It shows the local behaviour of the objects of a class. It presents abstractions of sets of possible states for objects, and which events change the state. A state is a function of the attributes values and the links of the object. An object can realize within a state some operations which do not modify its state, these operations are called activities or internal actions. An activity is an ongoing operation that takes time to complete whereas internal actions are instantaneously operations triggered by events. A change of state is called a transition and is triggered by an event which is something happening at a moment. An action can be connected to a transition, specifying what would be done in association with the state transition. As actions, events can be sent by objects to other objects. State diagrams are not drawn for all classes, only for those that have a number of well-defined states and where the behaviour is affected and changed by different events. They can be drawn for the system as a whole.

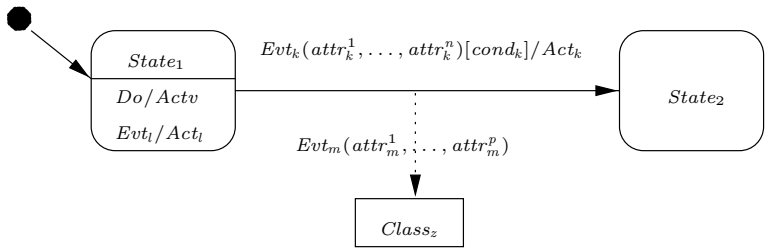


Fig. 2. A generic dynamic model

A state diagram is a directed graph whose nodes are states ($State_1$, $State_2$ in Fig. 2) and whose directed arcs are transitions. A transition is annotated by an event (Evt_k) and can optionally be followed by one or several attributes ($attr_x^y$) into brackets. A condition ($cond_k$) considered as a guard for the transition can be specified between hooks, actions (Act_k) are noted on the transition after the event

and the guard. Activities or internal actions are represented within the states. An activity ($Actv$) is noted after the keyword (Do) and internal actions (Act_l) follow the name of the associated event (Evt_l). When an object sends an event to another object, it is represented by an arrow in dotted lines that goes from the associated transition to a box denoting the name of the destination object class ($Classe_z$), the arrow is annotated by the event sent ($Evt_m(attr_m^1, ..., attr_m^p)$). Finally, the entry point in a state diagram is denoted by a full circle and an arrow toward the initial state.

3 From OMT Diagrams to a B Specification

This section presents the B method and a systematic transformation of the two previous OMT models into a B specification.

Created by Jean-Raymond Abrial, the B method [2] is a formal software development method that covers the software process from the specification to the implementation. B notations are based on the mathematical concepts of set theory, the language of generalised substitutions and the first order logic. Specifications are composed of abstract machines similar to modules or classes; they consist of a set of variables, properties on these variables called the invariant and operations. The state of the system, i.e. the set of variables values, is only modifiable by methods (operations). Various possibilities of machine links are given allowing large systems to be specified, and modular and reusable text to be produced. The B model is then refined, i.e. specialized, until a complete implementation of the software system is obtained. Refinement can be seen as an implementation technique but also as a gradually specification technique enabling progressive inclusion of details. In every stage of the specification, proof obligations are generated verifying that the operations preserve the invariant of the system. Refinement are proven correct by generation and proof of refinement proofs.

Note that, in this discussion, we don't take into account the evolution of B proposed by Abrial and Mussa in [1, 3].

3.1 From an Object Model to a B Specification

For each concept introduced in the section 2.1, we discuss its formalisation and define a B generic template. Then we build a B specification for the whole object model showing how the different components are distributed in various abstract machines.

Classes. Classes are expressed by abstract machines. If they are parameterized, names of formal parameters are expressed in brackets after the name of the machine and typed by a predicate inside the `CONSTRAINTS` clause. An abstract machine $\texttt{Class}_\texttt{i}$ describes a deferred set $\texttt{CLASS}_\texttt{i}$ of the possible instances of a class $Class_i$. The set of existing instances is modelled by a variable $\texttt{class}_\texttt{i}$ constrained to be a subset of $\texttt{CLASS}_\texttt{i}$. For each attribute $Attr_i$, a variable $\texttt{attr}_\texttt{i}$

is created and defined in the INVARIANT clause as a binary relation between the set $\mathtt{class_i}$ and its associated type $\mathtt{Type_{attr}}$. The formalisation of an attribute can be completed considering its different features. Table 1 shows how the binary relation can be defined more precisely depending on its existence (mandatory or optional) and its cardinality (multi or mono-valued).

Table 1. Formalisation of an attribute depending on its features

	Multi-valued	Mono-valued
Mandatory	$\mathtt{attr_i} \in \mathtt{class_i} \leftrightarrow \mathtt{Type_{attr}}$ $\wedge\ \mathtt{dom}(\mathtt{attr_i}) = \mathtt{class_i}$	$\mathtt{attr_i} \in \mathtt{class_i} \rightarrow \mathtt{Type_{attr}}$
Optional	$\mathtt{attr_i} \in \mathtt{class_i} \leftrightarrow \mathtt{Type_{attr}}$	$\mathtt{attr_i} \in \mathtt{class_i} \nrightarrow \mathtt{Type_{attr}}$

The template presented Fig. 3 shows a first specification for $Class_i$. This one will be completed later. Besides the definition of the class and its attributes, we introduce the initialisation of variables, standard instance creation or supression operations and a partial formalisation of a generic Op_i operation. The $\mathtt{create_i}$ operation creates an instance $\mathtt{new_i}$ of a class $\mathtt{Class_i}$ by choosing it in an indeterministic manner in $\mathtt{CLASS_i} \setminus \mathtt{class_i}$. The attribute $\mathtt{attr_i}$ is updated with its default value $\mathtt{val_{attr}}$. Note that, in case where there is no default value specified in the diagram, it must be given as a parameter. The $\mathtt{Op_i}$ operation is partially defined here and will be completed with the formalisation of the functional model.

The expression of the class multiplicity is immediate. It is formalised by adding a condition in the invariant which limits the cardinality of the $\mathtt{class_j}$ set :

$$\mathtt{card}(\mathtt{class_j}) \leq \mathtt{mult_j}$$

Relations. Associations between classes are identified by couples of instances. They are naturally expressed in B as binary relations between the existing instances of classes. The Ass_{ij} association is formalised by adding a variable $\mathtt{ass_{ij}}$ and a property of invariance defining it as a binary relation between $\mathtt{class_i}$ and $\mathtt{class_j}$ (see Fig. 4). When an association is represented by a class, attributes, associations and operations for this association are expressed as for classes and the referenced variable is the association one ($\mathtt{assClass_{ik}}$). Fig. 4 presents the B definition of the association $AssClass_{ik}$, its attributes $Attr_{ik} : Type_{ik}$ ($AttrList_{ik}$), its association Ass_{aj} with $Class_j$ and its operations Op_{ik} ($OperList_{ik}$). The role does not need a separate definition. It is calculated from the association and expressed in B by an alias in the DEFINITIONS clause. Fig. 4 shows how $Role_i$ and $Role_j$ are specified by two definitions ($\mathtt{role_i}$, $\mathtt{role_j}$) from the association Ass_{ij}. An association definition can be expressed more precisely according to the values of the role multiplicities. As shown in Table 2, properties can be deduced from the cardinality of a role (cardinality of $Role_i$ for Ass_{ij}). Once properties have

```
MACHINE Class_i(Param_i)
CONSTRAINTS
   Param_i ∈ Type_par
SETS
   CLASS_i
VARIABLES
   class_i ,
   attr_i
INVARIANT
   class_i ⊆ CLASS_i ∧
   attr_i ∈ class_i ↔ Type_attr
INITIALISATION
      class_i , attr_i := ∅, ∅
OPERATIONS
   oo_i < −create_i =
   PRE
      class_i ≠ CLASS_i
   THEN
      ANY new_i
      WHERE
         new_i ∈ CLASS_i \ class_i
      THEN
         class_i := class_i ∪ {new_i} ||
         attr_i := attr_i ∪ {(new_i ↦ val_attr)} ||
         oo_i := new_i
      END
END;
rmv_i(obj_i) =
PRE
   obj_i ∈ class_i
THEN
   class_i := class_i \ {new_i} ||
   attr_i := {obj_i} ⩤ attri
END;
s_i < −Op_i(obj_i, arg) =
PRE
   obj_i ∈ class_i ∧
   arg ∈ Type_arg
THEN
   ...
   s_i := ...
END;
```

Fig. 3. Template for a B specification of a class

been deduced for the two cardinalities, the association definition can be modified; we present as appendix in Table 4 possible translations of an association according to the most frequently used cardinalities.

The inheritance mechanism is expressed in the invariant by an inclusion predicate. For every subclass $SubClass_k$ of a superclass $Class_k$, a new abstract machine $\texttt{SubClass}_\texttt{k}$ is created. The translation process is the same as for a class except that there is no $\texttt{SUBCLASSE}_\texttt{k}$ enumerated set. Indeed, sets of all possible instances for the $Class_k$ class and the $SubClass_k$ subclass are the same. $\texttt{subClass}_\texttt{k}$ is then defined as a subset of $\texttt{class}_\texttt{k}$ (see Fig. 5).

Modelling the Whole Class Diagram into a Single B Specification. Specification modularity is an important aspect of the software development. B offers different assembly primitives in order to compose abstract machines. We will use here `USES` and `INCLUDES` clauses. They allow constants, sets, variables and properties of an abstract machine to be shared by another one. Moreover, the `INCLUDES` clause permits data modifications. We have previously presented the translation of each object model component. We will now show how these different components will be distributed in the different B abstract machines, merging them into a single specification.

Since a class encapsulates at the same time static and dynamic properties of a set of objects, it seems natural that all the constructs associated to a class are part of a same abstract machine.

```
MACHINE ...
   ...
VARIABLES
   ..., ass_ij,
   assClass_ik, attr_ik, ass_aj
DEFINITIONS
   role_i(x) == ass_ij[{x}] ;
   role_j(x) == ass_ij^-1[{x}]
INVARIANT
   ... ∧ ass_ij ∈ class_i ↔ classe_j ∧
   assClass_ik ∈ class_i ↔ class_k ∧
   attr_ik ∈ assClass_ik ↔ Type_ik ∧
   ass_aj ∈ assClass_ik ↔ class_j ∧
OPERATIONS
   setAss_ij(c_i, c_j) =
   PRE
      c_i ∈ class_i ∧ c_j ∈ class_j
   THEN
      ass_ij := ass_ij ∪ {(c_i ↦ c_j)}
   END;
   rmvAss_ij(c_i, c_j) =
   PRE
      c_i ∈ class_i ∧ c_j ∈ class_j ∧
      {(c_i ↦ c_j)} ∈ ass_ij
   THEN
      ass_ij := ass_ij \ {(c_i ↦ c_j)}
   END;
   s_ik < −Op_ik(obj_ik, ...) =
   PRE
      obj_ik ∈ assClass_ik ∧
      ...
```

Fig. 4. Template for a B specification of associations

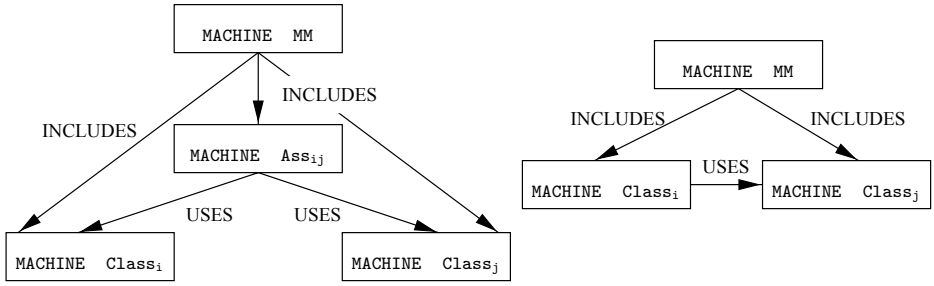
Table 2. Properties deduced from the multiplicity of roles

Multiplicities	Properties
0-1	$\mathtt{ass_{ij}} \in \mathtt{class_i} \nrightarrow \mathtt{class_j}$
0-x	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$ $\wedge \forall \mathtt{c_i}.(\mathtt{c_i} \in \mathtt{class_i} \Rightarrow \mathtt{card}(\mathtt{ass_{ij}}[\{\mathtt{c_i}\}]) \leq \mathtt{x})$
0 or more	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$
1	$\mathtt{ass_{ij}} \in \mathtt{class_i} \rightarrow \mathtt{class_j}$
1-x	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$ $\wedge \mathtt{dom}(\mathtt{ass_{ij}}) = \mathtt{class_i}$ $\wedge \forall \mathtt{c_i}.(\mathtt{c_i} \in \mathtt{class_i} \Rightarrow \mathtt{card}(\mathtt{ass_{ij}}[\{\mathtt{c_i}\}]) \leq \mathtt{x})$
1 or more	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$ $\wedge \mathtt{dom}(\mathtt{ass_{ij}}) = \mathtt{class_i}$
x-y	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$ $\wedge \mathtt{dom}(\mathtt{ass_{ij}}) = \mathtt{class_i}$ $\wedge \forall \mathtt{c_i}.(\mathtt{c_i} \in \mathtt{class_i} \Rightarrow \mathtt{x} \leq \mathtt{card}(\mathtt{ass_{ij}}[\{\mathtt{c_i}\}]) \leq \mathtt{y})$
x or more	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$ $\wedge \mathtt{dom}(\mathtt{ass_{ij}}) = \mathtt{class_i}$ $\wedge \forall \mathtt{c_i}.(\mathtt{c_i} \in \mathtt{class_i} \Rightarrow \mathtt{card}(\mathtt{ass_{ij}}[\{\mathtt{c_i}\}]) \geq \mathtt{x})$

```
MACHINE SubClass_k
VARIABLES
   ..., subClass_k, ...
INVARIANT
   ... ∧ subClass_k ⊆ class_k ∧ ...
```

Fig. 5. Template for the inheritance mechanism

Concerning association and inheritance, the problem is a little more complex. Lano [10] systematically includes the association concept inside one of the abstract machines which formalize a class participating in the association. Conversely, Nagui-Raiss [13] proposes to create a machine for each association. We adopt the solution developed in [9], consisting in differentiating fixed and unfixed associations. An association is fixed for a class if, for each instance of this class participating to the association, its link can be created only during the instance creation and never changed for its lifetime. An instance of association is thus created (removed) by the instance creation (suppression) operation of the class. To translate an Ass_{ij} association between $Class_i$ and $Class_j$, we apply the following rules: if Ass_{ij} has attributes or is not fixed for $Class_i$ and $Class_j$, an independent machine $\texttt{Ass}_{\texttt{ij}}$ is created. This machine uses machines $\texttt{Class}_{\texttt{i}}$ and $\texttt{Class}_{\texttt{j}}$ (see Fig. 6 (a)); if Ass_{ij} has no attributes or is fixed for $Class_i$ but not for $Class_j$, the constructs associated to the association are introduced into the $\texttt{Class}_{\texttt{i}}$ machine, this machine uses $\texttt{Class}_{\texttt{j}}$ (see Fig. 6 (b)). When Ass_{ij} is fixed for the two classes, there is no reason to choose one of them, the choice is given to the specifier. We have to introduce a new machine MM, because of a B restriction which constrains machines linked by an `USES` clause to be included (clause `INCLUDES`) in another one.



(a) Unfixed association

(b) Fixed association for $Class_i$

Fig. 6. Association representation

As for the association, several solutions have been proposed in the litterature for the inheritance translation. The adopted solution is the following: a subclass $SubClass_k$ `USES` a $Class_k$ superclass and a new machine $\texttt{inheritance}_{\texttt{k}}$ is created (see Fig. 7). The $\texttt{SubClass}_{\texttt{k}}$ machine allows the creation of $SubClass_k$ instances (but not as $Class_k$ instances) or the specialization of existing $Class_k$ instances as $SubClass_k$ instances. The machine $\texttt{inheritance}_{\texttt{k}}$ formalizes the whole hierarchy, it creates $SubClass_k$ instances as special cases of $Class_k$ instances by applying in parallel the creation operation of both classes.

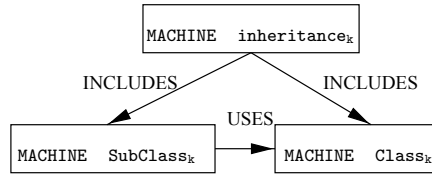


Fig. 7. Inheritance representation

Aggregation. The semantics of the aggregation in OMT is not very precise. Rumbaugh proposes to ignore the aggregation and stick to plain associations when you are not sure of its utility. We propose two ways of transforming the aggregation in B.

– We are in the case of a bottom-up construction of the specification. Components have been defined and we want to specify the compound from those. The compound is linked to its components but on the other hand components have a proper existance and can be used for the definition of other concepts. Aggregation is then regarded as a special association (see Fig. 8) and the previously template for the association is applied to generate the B specification.

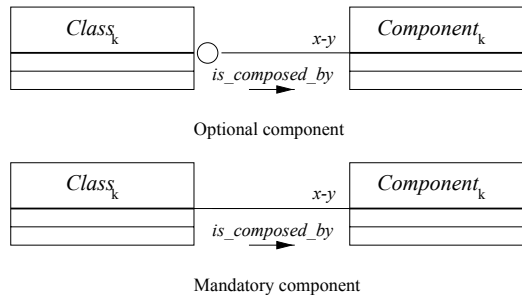


Fig. 8. Aggregations as associations

– We are in the case of a top-down construction of the specification. The complexity of a class definition is decreased by breaking it down into several components. Here, components do not have any proper existence. The translation of the aggregation is then processed using the template presented in Fig. 9. The compound machine `INCLUDES` the component machine. For each aggregation, a variable `comp` represents the aggregation by a binary relation between the set of existing instances of the compound and the set of existing instances of the component. This relation is refined by using Table 4, tak-

ing into account C_i as the multiplicity expressed in the OMT diagram (x-y) and C_j as `1` wether the component is mandatory or `0-1` if the compound is optional.

```
MACHINE  Component_k
SETS
   COMPONENT_k
VARIABLES
   component_k, ...
INVARIANT
   component_k ⊆ COMPONENT_k ∧
   ...
```

```
MACHINE  Class_k
INCLUDES  Component_k
SETS
   CLASS_k
VARIABLES
   class_k, comp, ...
INVARIANT
   class_k ⊆ CLASS_k ∧
   comp ∈ class_k ↔ component_k
   ...
```

Fig. 9. Template for the aggregation

Constraints. Constraints expressed in the object model are translated by adding conditions to the invariant of the various abstract machines. Constraints concerning attributes of a class will be added in the abstract machine which models this class. Those which treat only with associations are added in the machine symbolizing the association. If constraints combine both type of data, they are added to the invariant of a machine which includes both machines (typically the `machine MM` introduced in Fig. 6).

Thereafter, the single specification resulting of the object model translation is called `O-SPEC`.

3.2 Taking into Account Components of a Dynamic Model

For each concept introduced in the section 2.2, we discuss its formalization and present the generated B specification. Then, we take into account these components and integrate them into the `O-SPEC` specification resulting from the object model translation, giving a single specification.

States. For each diagram associated to the class *Class*$_i$, we create an enumerated set `STATE_i` which gathers all the states of the diagram. The state of an object is recorded by a variable `state_i` defined as a function from the set `class_i` of existing instances of *Class*$_i$ to `STATE_i`. Thus, the state of an object `oo_i` is given by the result of `state_i(oo_i)`. Changing the state corresponds to the modification of this function. The initial state is set up in the instance creation operation as it is done for class attributes. Fig. 10 presents the formalization of the states of the generic dynamic model (assigned to *Class*$_i$) presented in Fig. 2.

SETS
$\quad \mathtt{STATE_i} = \{\mathtt{state_1}, \mathtt{state_2}\}$
VARIABLES
$\quad \mathtt{state_i}, \ldots$
INVARIANT
$\quad \mathtt{state_i} \in \mathtt{class_i} \rightarrow \mathtt{STATE_i} \wedge$

Fig. 10. Template for the state

An alternative state representation is proposed by Lano [10] in which each state of the diagram is represented by a subset of an existing instances set. Modification of a state is processed by moving the objects from one state set to the other. This approach has the advantage to generate proof obligations which are more simple and easier to resolve, but it becomes infeasable once the number of states is important.

Transitions. Each event is formalized by an operation. This operation is parameterized by the target objects and the eventual parameters of the event. Parameters are typed by a predicate in the precondition of the operation. The operation is defined by a `SELECT` substitution which has as many cases as transitions where the event appears. The operation modifies the state of the object and calls the operations associated to actions and events specified in the transition. As presented below, an action is formalized by a B operation, this operation results from the object model and will be completed by the functional model translation. Actitivities are extra actions which take care that the object is in the appropriate state.

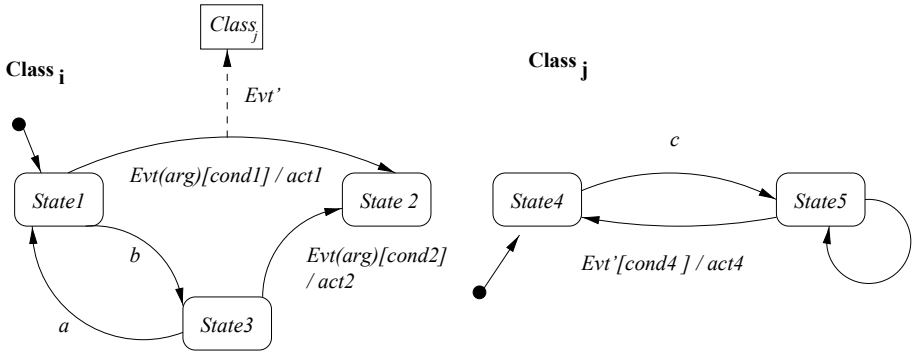


Fig. 11. State diagrams for $Class_i$ and $Class_j$

The translation of an event `Evt` used in the $Class_i$ state diagram (see fig. 11) is presented in Fig. 12.

```
Evt(c_i, arg) =                    |     SELECT  state_i(c_i) = state1 ∧ cond1
PRE                                |        THEN  state_i(c_i) := state2 || act1 || Evt'
   c_i ∈ class_i ∧                 |     WHEN  state_i(c_i) = state3 ∧ cond2
   arg ∈ Type_arg                  |        THEN  state_i(c_i) := state2 || act2
THEN                               |     ELSE skip END
                                   |  END
```

Fig. 12. Template for an event

Integrating the Dynamic Model into the O-SPEC Specification. We have presented the formalization of each concept of the dynamic model. This section is devoted to the integration of these concepts into a whole specification, taking into account that an object model has been first defined, formalised with the O-SPEC specification.

The B formalization of states of a class $Class_i$ must be defined in the abstract machine associated to this class. We consider that a state is a data of a class and therefore must be formalized in the abstract machine defining this class.

When translating transitions, several concepts have been identified. We present now separately each of them, and express the needed conditions.

- Target objects: they are instances of different classes. These instances are reached via the set of existing instances (variable $\mathtt{class}_i$ for the class $Class_i$). Thus, the machines where these various variables are defined must be accessible in the machine where we formalize target objects.
- Firing conditions: they are predicates on attributes and associations of a class. We saw previously that attributes and associations of a class can be specified in different machines (not fixed association). The firing condition has to be introduced in a machine which reaches data of both others.
- Action calls: they are operations which modify the data of a class or an association and are formalized by translation of object and functional models. We distinguish between the actions which modify attributes of a class and those which modify associations. The first ones are specified in the machine representing the class and the second ones are defined in the machine modelling the association. Here, we are constrained by several obligations: the necessity to include machines where actions are defined and if we are in the machine which defines the action, the necessity to rewrite its definition in the body of the operation which calls it.
- Access and modification of states: states are formalized by variables defined as a total function from a set of existing instances to an enumerated set of all the possible states. The access to a variable is possible outside the machine where it is defined whether it belongs to the set of included or used machines. The modification of a variable is authorized outside the machine A where it is defined only by the use of its modification operations. The machine A must then be included.
- Communication with other events: sending an event by another one is expressed by an operation call. The constraints inherent in the event call are

the same ones as for actions. If the called operation is in the same machine than the calling operation, the text of the called one must be rewritten in the calling one, if not the machine specifying the called operation must be included in the machine of the calling one.

The constraints to be respected being now clearly established, we present our proposal for the translation of the event and associated constructs. This proposition is illustrated in Fig. 13 which takes again the translation of the event `Evt` :

- Each transition gives rise to an operation (`TransitionSt1St2`, ...) in the machine where the state is defined ($\texttt{Class}_\texttt{i}$). This operation modifies the state value ($\texttt{state}_\texttt{i}(\texttt{c}_\texttt{i})$) and calls actions (`act1`) which change the attributes of the above class. It is parameterized by the target object ($\texttt{c}_\texttt{i}$) and constrained by a precondition specifying the source state (`state1`) of the transition.
- Each event gives rise to an operation (`Evt(...)`) in a new machine (`Control`) which includes the machines ($\texttt{Class}_\texttt{i}, \texttt{Class}_\texttt{j}$) in which data used by this event are defined. This operation is defined as seen in the previous section. However, some modifications were brought in the body : the state modification and actions associated to a class are replaced by calling the transition operation (`TransitionSt1St2`, ...), it remains the call of action modifying the association and the call of event operation (`Evt'`).

Care has to be taken when calling operations in B : an operation of a machine cannot be called from another operation of the same machine. This can be solved with the specification of intermediate definitions (`Defact1, ...`) and the use of those instead of the operation. This solution has several benefits :

- the text of the operation is specified only once (`act1, act2, act3`);
- no cycles between operations can be introduced;
- a distinction between internal and external events can be simulated. Each internal event (`Evt'`) is specified by a definition (`DefEvt'`) and has no own operation. External events (`Evt`) are defined as operations in which internal events can be used by way of the B definitions. The concept of public (traditional in B) and private operation is added here by means of the definitions.

Conditions can be added when analysing the different states. A state is a function of the values of the attributes and the links of this object. The state of an object is also function of the states of its components when the corresponding class is built from an aggregation. Suppose that a property P_i expresses a state $State_i$ from the values of the attributes, links and components states of the $Class_i$ class. Then, a new constraint must be added in the invariant clause of the machine which includes machines formalizing class, links and components. This condition binds the state $State_i$ and the property P_i ensuring that the system is always in a coherent state (see the invariant of the `Control` machine, Fig. 13).

In short, our proposal specifies the data and the operation of modification in an abstract machine. This machine symbolizes either a class or an association.

```
MACHINE Class_i
SETS
   STATE_i = {state1, state2, state3};
   ...
VARIABLES
   ..., state_i
INVARIANT
   ... ∧ state_i ∈ class_i → STATE_i
DEFINITIONS
   Defact1 == ...;
   Defact2 == ...
OPERATIONS
   act1 = Defact1 ;
   act2 = Defact2 ;
   TransitionSt1St2(c_i) =
   PRE
      c_i ∈ class_i ∧
      state_i(c_i) = state1
   THEN
      state_i(c_i) := state2 ||
      Defact1
   END;
   TransitionSt3St2(c_i) =
   PRE
      c_i ∈ class_i ∧
      state_i(c_i) = state3
    THEN
      state_i(c_i) := state2 ||
      Defact2
   END;
      ...
MACHINE Class_j
SETS
   STATE_j = {state4, state5}; ...
VARIABLES
   ..., state_j
INVARIANT
   ... ∧ state_j ∈ class_j → STATE_j
 DEFINITIONS
   Defact3 == ...; ...
OPERATIONS
   act3 = Defact3 ;
   TransitionSt4St5(c_j) =
   PRE
      c_j ∈ class_j ∧ state_j(c_j) = state4
   THEN
      state_j(c_j) := state5 ||
      Defact3
   END;
      ...
MACHINE Control
INCLUDES
   Class_i, Class_j
INVARIANT
   ∀ c_i.(c_i ∈ class_i ⇒
      (state_i(c_i) = state1 ⇔ P_i(c_i)))   ∧
   ...
DEFINITIONS
   DefEvt' == ...; ...
OPERATIONS
   Evt(c_i, c_j, arg) =
   PRE
      c_i ∈ class_i ∧ c_j ∈ class_j
      ∧arg ∈ Type_arg
   THEN
      SELECT state_i(c_i) = state1 ∧ cond1
         THEN TransitionSt1St2(c_i)
         || DefEvt'
      WHEN state_i(c_i) = state3 ∧ cond2
         THEN TransitionSt3St2(c_i)
      ELSE skip END
   END
```

Fig. 13. Integration of the dynamic model in the O-SPEC specification

The logical sequence of the operation calls, the synchronization of the various events and actions are defined in one or more other machines (`Control`) which include the preceding machines. When state constraints must be added, they have to be specified in the `Control` machine.

4 Advantages of This Approach

The advantages of semi-formal methods are known, they facilitate the analysis and the construction of the software. In this section, we study repercussions of our approach in the proof process.

We start by specifying two generic models (see Fig. 14): an object and a dynamic one. These models gather the principal components of the OMT notation and are expressed here as general as possible.

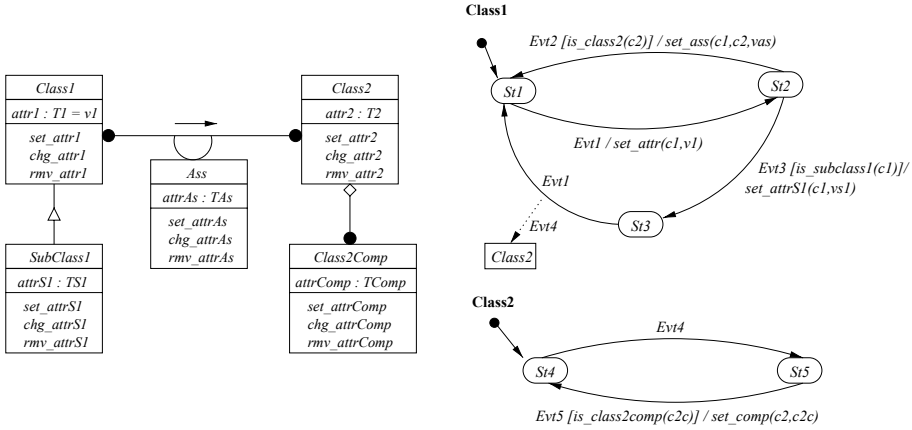


Fig. 14. Generic OMT models

The object model presents *Class1* and *Class2* classes linked by an *Ass* association. The *Class2* class is aggregated by the *Class2Comp* optional component. A *SubClass1* subclass inherits from *Class1*. Attributes and operations are defined for each class and association. The dynamic model is expressed by two state diagrams associated to *Class1* and *Class2* classes. Objects of *Class1* have three possible states (*St1, St2, St3)*, the change of the state is triggered by *Evt1, Evt2* and *Evt3* events. When the *Evt1* event occurs in the *St3* state, it sends to *Class2* an *Evt4* event. *Evt3* and *Evt4* are internal events. Actions are associated to transitions, they modify the data of the *Class1* class, the *Ass* association or the *SubClass1* subclass. The state diagram associated to *Class2* is composed of two states (*St4, St5*), two events *Evt4, Evt5* and an action which updates the aggregation.

We apply our transformation templates on OMT models, obtaining a B specification (`SPEC-B`), the structure of which is represented graphically in Fig. 15.

The correction of a B specification is done by generating and proving proof obligations. These proof obligations verify that operations preserve the invariant of the system. We thus generate the proof obligations for `SPEC-B` using the B tool [19]; 236 proof obligations were generated, they were all automatically proven by the B prover as shown in table 3.

The generated `SPEC-B` specification is automatically proved within the B Tool. It is proved relatively to the invariant generated by the transformation of the generic specification.

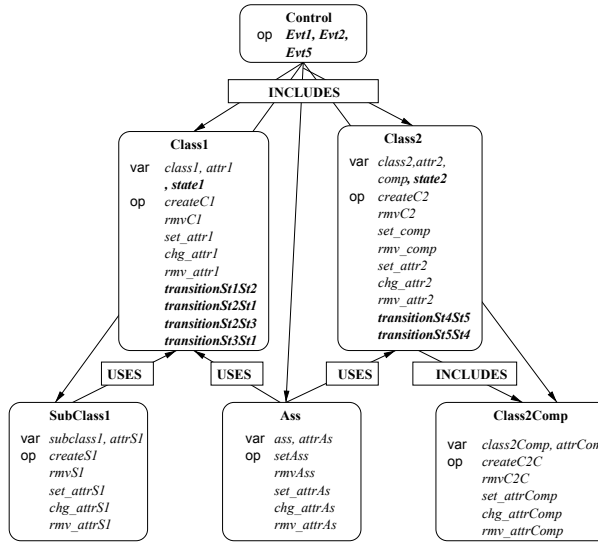


Fig. 15. Structure of the SPEC-B specification

Table 3. Proof results

COMPONENT	Obv	nPO	nUn	%Pr
Ass	5	6	0	100
Class1	28	23	0	100
Class2	71	34	0	100
Class2Comp	10	8	0	100
Control	37	3	0	100
SubClass1	5	6	0	100
TOTAL	156	80	0	100

The specification is modular since the used templates facilitate its decomposition into a set of B machines. The proof is then decomposed. When specifying a real problem, we have to instantiate these templates. The part corresponding to the predefined templates is automatically proved. The creative work of the specification will add new proof obligations linked to constraints between components corresponding to the problem and its domain knowledge. The proof of this creative part remains to be done.

5 Related Work

Many studies have proposed to combine semi-formal and formal methods. Several projects have studied the problem of integrating SSADM and Z [14], OMT and LOTOS [20] - VDM [7] or Z [5, 6].

Next authors have addressed the more specific translation from OMT semi-formal specifications into B specifications :

- Nagui-Raiss [13] proposes rules from extended entity association diagrams to B.
- Shore [18] shows how the different structures of object-oriented analysis can be represented in B and the limitations introduced by the B language. He proposes to use those transformations especially for the implementation.
- Facon and all [9, 8] defines rules from both OMT static and dynamic models to B specification. They are interested in modularizing the formal specification resulting from the object model transformation.
- Lano [10, 11] illustrates the use of object-oriented method for the development process in B. He proposes several processes to map analysis models expressed in OMT notations into B machines.
- Sekerinsky [17] translates Statecharts in B.

But all these works differ from our approach in some of the following aspects :

- They do not deal with the aggregation and contraints in the object model.
- Most studies do not differentiate between the various cases of attributes or associations. We give here a systematic transformation of attributes depending on their features and of associations depending on their cardinalities.
- Consistency between the models is not treated. Our proposition integrates the dynamic model into the B specification resulting from the object model translation, leading to a single specification.
- Most studies have not investigated all the advantages of this approach : we show here how this approach improves the proof process.
- This paper presents the transformation using generic templates, taking particular care to remain the most generic and general as possible.

6 Conclusion

In this paper, we have presented an intuitive semantic for the set of concepts used in the OMT object and dynamic models. A B generic template has been defined for each component as well as their integration into a single specification. Rules have been explicited to merge informations coming from both object and dynamic OMT models, providing an integrated view of dynamic and static requirements. Advantages for using these templates are :

- the translation from OMT object and dynamic diagrams into a B specification is done in a systematic way,
- in the formal specification, the two object and dynamic OMT models are combined giving a single specification,
- a proof of the invariant preservation for the resulting B specification is automatically done by the B-tool.

Currently, in our approach we consider B and OMT as two idependant formalisms, regarding those as two views of the specification construction and taking the advantages of each one.

A number of features of the OMT notations have been left out. These remain the subject of further research. We currently study refinement, with the idea of applying it either on the OMT models or on the B specification. An idea is to use the concepts of sub-systems, composite-objects and composite states introduced in the OMT notations and see if they can be related to the B refinement.

Tools are expected in order to automatically transform the OMT diagrams into a B specification. We aim to integrate the templates defined below as operators in the PROPLANE model [12]. The PROPLANE approach is related to process modelling, methodological aspects are expressed outside the language by way of development operators. Operators are applied to development steps which are composed of both a workplan describing the followed reasoning and the specification.

References

[1] J.R. Abrial. Extending B without Changing it (for Developing Distributed Systems). In H. Habrias, editor, *Putting Into Practice Methods and Tools for Information System Design - 1st Conference on the B Method*, Nantes (F), November 1996.

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

[3] J.R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method - 2nd International B Conference*, number 1393 in Lecture Notes in Computer Science, Montpellier (F), April 1998. Springer-Verlag.

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

[5] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Integrating OMT and Object-Z. In *BCS FACS/EROS ROOM Workshop*, London (UK), June 1997.

[6] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Translating the OMT Dynamic Model into Object-Z*. In *ZUM'98: The Z Formal Specification Notation - 11th International Conference of Z Users*, number 1493 in Lecture Notes in Computer Science, Berlin (D), September 1998.

[7] P. Facon and R. Laleau. Des modèles d'objets aux spécifications formelles ensemblistes. *Revue Ingénierie des Systèmes d'Information*, 4(2), 1996.

[8] P. Facon, R. Laleau, and H.P. Nguyen. Dérivation de spécifications formelles B à partir de spécifications semi-formelles de systèmes d'information. In H. Habrias, editor, *Putting Into Practice Methods and Tools for Information System Design - 1st Conference on the B Method*, Nantes (F), November 1996.

[9] P. Facon, R. Laleau, and H.P. Nguyen. Mapping Object Diagrams into B Specifications. In A. Bryant and L. Semmens, editors, *Methods Integration Workshop*, Electronic Workshops in Computing (eWiC), Leeds (UK), March 1996. Springer-Verlag.

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

[11] K. Lano, H. Haughton, and P. Wheeler. *Formal Methods and Object Technology*, chapter Integrating Formal and Structured Methods in Object-Oriented System Development, pages 113–157. FACIT. Springer-Verlag, 1996. S.J. Goldsack and S.J.H. Kent eds.

[12] N. Lévy and J. Souquières. Modelling Specification Construction by Successive Approximations. In M. Johnson, editor, *6th International AMAST Conference*, pages 351–364, Sydney (A), 1997. Springer Verlag 1349.

[13] N. Nagui-Raiss. A formal Software Specification Tool Using The Entity-Relationship Model. In *13th International Conference on the Entity-Relationship Approach*, number 881 in Lecture Notes in Computer Science, Manchester (UK), December 1994. Springer-Verlag.

[14] F. Polack, M. Whiston, and K. Pronk. The SAZ Project: Integration SSADM and Z. In *International Symposium Formal Methods Europe*, number 670 in Lecture Notes in Computer Science, Odense (DK), April 1993. Springer-Verlag.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall Inc. Englewood Cliffs, 1991.

[16] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Guide*. Addison-Wesley, 1998. ISBN 020130998X.

[17] E. Sekerinski. Graphical Design of Reactive Systems. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method - 2nd International B Conference*, number 1393 in Lecture Notes in Computer Science, Montpellier (F), April 1998. Springer-Verlag.

[18] R. Shore. An Object-Oriented Approach to B. In H. Habrias, editor, *Putting Into Practice Methods and Tools for Information System Design - 1st Conference on the B Method*, Nantes (F), November 1996.

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

[20] E. Wand, H. Richter, and B. Cheng. Formalizing and integrating the dynamic model within OMT. In *19th International Conference on Software Engineering*, Boston (USA), July 1997.

[21] P.T. Ward. How to Integrate Object Orientation with Structured Analysis and Design. In *IEEE Software*. March 1989.

Appendix

Table 4. Association (Ass_{ij}) translations according to the most frequently used cardinalities (C_i, C_j)

		$\mathtt{c_j}$			
		0-1	1	0 or more	1 or more
$\mathtt{c_i}$	0-1	$\mathtt{ass_{ij}} \in \mathtt{class_i}$ ⤔ $\mathtt{class_j}$	$\mathtt{ass_{ij}} \in \mathtt{class_i}$ ⤗ $\mathtt{class_j}$	$\mathtt{ass_{ij}} \in \mathtt{class_i} \nrightarrow \mathtt{class_j}$	$\mathtt{ass_{ij}} \in \mathtt{class_i}$ ⤀ $\mathtt{class_j}$
	1	$\mathtt{ass_{ij}} \in \mathtt{class_i} \rightarrowtail \mathtt{class_j}$	$\mathtt{ass_{ij}} \in \mathtt{class_i}$ ⤖ $\mathtt{class_j}$	$\mathtt{ass_{ij}} \in \mathtt{class_i} \rightarrow \mathtt{class_j}$	$\mathtt{ass_{ij}} \in \mathtt{class_i} \twoheadrightarrow \mathtt{class_j}$
	0 or more	$\mathtt{ass_{ij}}^{-1} \in \mathtt{class_j} \nrightarrow \mathtt{class_i}$	$\mathtt{ass_{ij}}^{-1} \in \mathtt{class_j} \rightarrow \mathtt{class_i}$	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$ $\wedge \mathtt{ran}(\mathtt{ass_{ij}}) = \mathtt{class_j}$
	1 or more	$\mathtt{ass_{ij}}^{-1} \in \mathtt{class_j}$ ⤀ $\mathtt{class_i}$	$\mathtt{ass_{ij}}^{-1} \in \mathtt{class_j} \twoheadrightarrow \mathtt{class_i}$	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$ $\wedge \mathtt{dom}(\mathtt{ass_{ij}}) = \mathtt{class_i}$	$\mathtt{ass_{ij}} \in \mathtt{class_i} \leftrightarrow \mathtt{class_j}$ $\wedge \mathtt{dom}(\mathtt{ass_{ij}}) = \mathtt{class_i}$ $\wedge \mathtt{ran}(\mathtt{ass_{ij}}) = \mathtt{class_j}$