Ensuring specification correctness by construction

Dieu Donné Okalas Ossami, Jeanine Souquières, Jean-Pierre Jacquot

Ensuring specification correctness by construction

Dieu Donné Okalas Ossami, Jeanine Souquières, Jean-Pierre Jacquot

LORIA - Université Nancy 2 - UHP Nancy 1
Campus scientifique, BP 239
54506 Vandœuvre-lès-Nancy Cedex - France
Email: {okalas,souquier,jacquot}@loria.fr

Abstract

We propose a process model for the development of formal and semi-formal specifications based on the notions of multi-view states and development operators. A specification state is composed of a UML and a B view. The development of a specification is seen as a sequence of application of operators, which model design decisions and make both views evolve. To produce consistent specifications, we define a consistency relation between views, allowing to define and check operators' correctness. Thus, the development process guarantees that the specification can be safely verified.

Keywords: consistency, correctness, verification, validation, operator, development process, multi-view, UML, B.

1 Introduction

Experience has shown that the most critical and least supported phases of the software life cycle are requirements analysis and specification. Errors and misconceptions in the requirements will be passed on the system specifications and from them down the process to show up ultimately in the programs. Formal specifications could greatly help in reducing the amount of errors because of the absence of ambiguity in formal texts and the availability of powerful analysis techniques and prototyping tools. However, formal specifications are hard to write and, more importantly, hard to read; this raises the problem of the validation of the specification. We believe that the effective availability of tools supporting specification development could greatly help in promoting the use of formal specifications by practitioners. Tool support should include guidance during the specification development process; it should enable users to develop specifications in an intuitive fashion by separating the use of design concepts from the technical details of how they are captured in specification languages. The specification development process should be problem oriented instead of language oriented.

Validation requires users of the system to be able to "read" the specification, hence the importance of graphical notations. Verification requires a formal notation. The current issue is that no single language offers both kinds of notation. Is it possible to combine graphical notations and formal languages ? Currently, there are two mains streams of specification languages: graphical notations such as UML [RJB98] and mathematical notations such as B [Abr96], Z [Spi92], etc.. Our goal is to design a framework where both kinds of notations can be used together to fulfill the needs of all the people involved. Our approach aims at capitalising on existing languages rather than at defining a new one. This allows us to reuse the efforts that have been out in the production of industrial tools such as Rational Rose [1] or ArgoUML [2] for the edition of UML diagrams, and such as *l'Atelier B* [STE98], B-Toolkit[BCL96], or b4Free [B4F] for the formal verification of specifications. Our framework supports multi-view specification activity by providing assistance during the development process. Its key is the notion of development operators : the development of a specification is defined as a sequence of steps, each of which maps a development state to the next by the application of an operator.

The formalisation of object-oriented concepts has prompted many research works. Three general approaches are identified in the literature: (1) extension of formal notations with object-oriented concepts, (2) extension of object-oriented notations with formal notations, and (3) method integration between object-oriented and formal notations. Z++ [Lan91] and Object-Z [CDD$^+$90] are examples of the first approach where Z [Spi92] is supplemented with object-oriented concepts and notations. In the second approach, parts of the informal specifications expressed in natural language are replaced by formal statements expressed in a well-known formal language, e.g. Syntropy [CD94]. In the third approach, transformation rules are defined which translate specification written in one formalism into an "equivalent" specification written in another formalism. One instance of this approach is UML to B transformation: it allows specifiers to use formal techniques and tools to check the specification. Transformation provides us with automated support to generate a B specification from UML diagrams [LP01,MS99,LS02,SBO03] taking into account OCL constraints [LS02,ML02]. Another instance is B to UML transformation: it eases the validation by the generation of UML diagrams (class diagrams and state diagrams) from a B specification [HM04,IL04,TV03].

One major problem in UML and B integration approaches is maintaining the consistency when the specification evolves. Currently, UML and B integration approaches offer either UML to B [LSC03,SB] or B to UML transformations [TV01] but not both in the same tool. Several reasons account for this state of affair, but the net result is the practical impossibility to define a process where both kinds

[1] http://www-306.ibm.com/software/rational
[2] http://www.argouml.tigris.org

of transformation can be symmetrically used. As a consequence, UML to B or B to UML transformation induces a sequential development process where : (i) a new specification in the target formalism is generated each time the rules are used. Thus, any information added in the generated specification is lost and must be redesigned; (ii) the modifications brought into the generated specification cannot be retrofitted. This raises the issue of consistency between the current B specification and its corresponding UML specification [LP02].

The paper is organised as follows. Section 2 presents the approach with a definition of the consistency relation between two developments steps to ensure the correctness of the construction. Section 3 presents a selection of development steps on the generalised railroad crossing case study using operators. Section 4 describes how the correctness of an operator can be verified. This is based on the verification of the consistency relation on obtained specifications when applying that operator. Section 5 concludes the paper.

2 Operators' framework : a general description

Our approach aims at modelling a process for developing specifications expressed simultaneously in an object oriented notation graphical (UML) and in a formal notation (B). Both specifications are built by successive approximations. Operators are the central notion: they capture strategies and design concepts. They enable the user to develop specifications in an intuitive fashion by separating the use of design concepts from the technical details of how they are captured in the chosen specification languages. Different development strategies can be modelled as libraries of operators, allowing to provide users with flexible development processes.

2.1 Specification state and operator

Our process model is strongly inspired by the transformation approaches. The final specification results from a sequence of applications of transformers: *operators*. An operator is applied to a *specification state* and produces a new specification state.
A specification state consists of two views. The UML view provides users with a graphical notation and gives access to validation tools. The B view provides users with a formal notation and gives access to verification tools. The fundamental point is that the views are *two different expressions* of the *same specification*. A state is noted as:

$$Spec = \langle\, SpecUML,\ SpecB \rangle$$

A development operator transforms simultaneously the UML and the B views [OSJ05]. Often, the application of an operator requires some input from the user, the parameters. An operator consists of :

- application conditions, which is a predicate on the current development state.

- a description of the actions performed on *SpecUML* and *SpecB*, denoted by $\mathcal{O}_{UML}$ and $\mathcal{O}_B$ and

2.2 Consistency relation

Our development model is based on the idea that applying a “correct” operator on a “correct” state leads to a new “correct” state. The question is now to define what “correct” means precisely. In previous work [OSJ05] we defined a so-called *consistency relation* between UML and B specifications, denoted by $\mathcal{R}_{el_C}$. It is defined as a conjunction of four conditions which are formally expressed in Definition 2.1. Let :

- $\mathbb{T}_{U\to B}$ be the set of UML to B transformation rules [LS01,MS99] which associate each UML artifact with one or more B artifacts.
- $\mathbb{T}_{U\to B}(SpecUML)$ be the application of $\mathbb{T}_{U\to B}$ on *SpecUML*.
- $SpecUML_{|\mathbb{T}_{U\to B}}$ be the restriction of *SpecUML* to elements for which there is a transformation rule to B defined in $\mathbb{T}_{U\to B}$.
- $ID(SpecB)$ and $ID(SpecUML_{|\mathbb{T}_{U\to B}})$ be sets of identifiers appearing in *SpecB* and in $SpecUML_{|\mathbb{T}_{U\to B}}$, respectivelly.

Definition 2.1 (Formal definition of the consistency relation)

$SpecUML\ \mathcal{R}el_C\ SpecB$:

(1) $\mathcal{WF}(SpecUML)\ \wedge\ \mathcal{WF}(SpecB)$

(2) $consistent(SpecUML)\ \wedge\ consistent(SpecB)$

(3) $\forall\ e_U.(e_U \in ID(SpecUML_{|\mathbb{T}_{U\to B}}) \Rightarrow$
$\exists\ \{e_B\},\ T.(\{e_B\} \subseteq ID(SpecB)\ \wedge\ T \in \mathbb{T}_{U\to B}\ \wedge\ T(e_U) = \{e_B\}))$

(4) $\forall\ \phi.(\mathbb{T}_{U\to B}(SpecUML) \vDash \phi \Rightarrow SpecB \vDash \phi))$

1 **Syntactic conformance.** It states that both *SpecUML* and *SpecB* must be well-formed. It ensures that the specification conforms to abstract syntax specified by the meta-model, i.e. UML meta-model or B abstract syntax tree. Let $\mathcal{WF}(SpecUML)$ and $\mathcal{WF}(SpecB)$ be two predicates defining if a UML and a B specification are well-formed.

2 **Local consistency.** It requires that both specifications must be internally consistent. That means they do not contain contradictions, but they could be incompletely defined. We write it $consistent(SpecUML)$ and $consistent(SpecB)$.

3 **Elements traceability.** It states that for any elements of $ID(SpecUML)$, e_U, that can be transformed by a rule T, there exists in $ID(SpecB)$ a set of artifacts $\{e_B\}$ resulting from the application of T to e_U.

4 **Semantic preservation.** It states that any statement ϕ satisfying the semantics of *SpecUML* must satisfy *SpecB*. The semantics of *SpecUML* is defined as $\mathbb{T}_{U\to B}$ $(SpecUML)$. This means that UML artifacts that have no B semantics defined in $\mathbb{T}_{U\to B}$ are not concerned by the consistency relation $\mathcal{R}el_C$. This has important

implications throughout the verification process. For example, it is well known that checking pairwise integration of a set of software specifications is only possible if one is able to transform them into a semantic domain supported by tools. B is our semantic domain and any UML statement that has no B formalisation cannot be verified in our framework.

We use the B theorem prover to prove that a statement ϕ holds in *SpecB* (condition (2)) and due to $condition(3)$, we derive the consistency of *SpecUML*, and therefore the consistency of the multi-view specification *Spec*.

2.3 Operator correctness

Given a specification state, $\langle \mathit{SpecUML}, \mathit{SpecB} \rangle$, a chosen operator, *Operator* with its parameters, *param*, the goal is to check that the new specification state, $\langle \mathit{SpecUML}', \mathit{SpecB}' \rangle$, obtained by the application of *Operator* is consistent, see Fig 1.

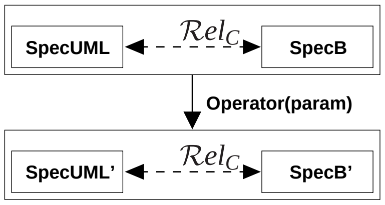


Fig. 1Correctness of an operator

The correctness of an operator is defined by means of a formula of the form $H \Rightarrow G$, where H denotes the hypothesis on the current development state and G the goal to be demonstrated based on the obtained development state according to the consistency relation.

(i) **Hypothesis**
- SpecUML $\mathcal{R}el_C$ SpecB : the current state of the development, satisfying the consistency relation,
- ApCond : the application conditions of the applied operator.

(ii) **Goal**
- SpecUML' $\mathcal{R}el_C$ SpecB'
  where :
  $\langle$SpecUML', SpecB'$\rangle$ = $Operator(param)$, whereby $\langle$SpecUML, SpecB$\rangle$ is an implicit parameter.

The proof obligation associated to *Operator* assuming the hypothesis is expressed as follows :

$$\boxed{\text{SpecUML } \mathcal{R}el_C \text{ SpecB} \wedge \text{ApCond} \Rightarrow \text{SpecUML' } \mathcal{R}el_C \text{ SpecB'}}$$

3 A small case study

Let us consider the generalised railroad crossing (GRC) case study [JS00]. The system to be specified aims at controlling a gate at a railroad crossing so that trains can safely go through. The informal text describes the problem as a monitoring of trains. The GRC lies in a region of interest *R*, as presented in Fig. 2. Trains travel in one direction through *R*, which is decomposed into three regions : *far, near* and *on*. The regions determine the position of trains in *R*.

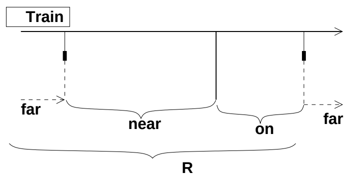


Fig. 2 The generalised railroad crossing

We will present three development steps to illustrate our approach, starting from this informal description. For each step, we give the idea we follow, the operator chosen in the library with its parameters and the new state produced by the application of the operator on the current state of the specification. In the new specification state, the new UML part is written in bold face and the new B part is written in a box. For space reasons, we do not give the formal definition of all operators used in this paper. Only the formal definition of the *Model-StateMachine* operator is given in the appendix (cf. section 6). The generic template for describing operators has been discussed in [OSJ05].

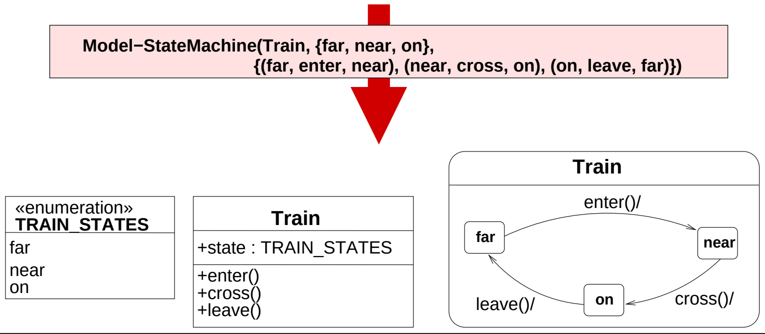
3.1 *First development step : modelling the state dependent behaviour of the train*

From the informal requirement, we identify three states (*far*, *near*, *on*) and three events (*enter*, *cross*, *leave*) which change the state of the train when it arrives, crosses or leaves the region *R*. This leads us to use the specification technique of introducing a state machine to model the description. This technique is captured by the development operator, *Model-StateMachine*. The required parameters are extracted from the text description and the application of the operator leads to the development state presented Fig. 3. The resulting UML view is composed of a class diagram with one class (*Train*) and an enumerated type, *TRAIN_STATES*, and of a state-transition diagram. Three machines and a refinement have been introduced in the B view.

- The *Train* machine corresponds to the class *Train*. It introduces a variable, *train* that specifies current objects of *Train*. The state of an object is recorded by the variable *Train_state* of type *TRAIN_STATES* which gathers all the states as specified in the corresponding state diagram. *Train_state* is defined as a function from *train* to *TRAIN_STATES*. Thus, the state of an object *oo* is defined as *Train_state(oo)*. Transitions between states are formalised by B operations which model the change of the state : *Train_TransFarNear* models the transition from the state *far* to the state *near*.
- Since events can affect data of several classes, B operations for events are modelled in the *System* machine which includes the *Train* machine. The *System* machine simulates the execution of the state diagram. However, since B does not allows sequencing in abstract machines, operations in the *System* machine are refined in the refinement *System_ref* in order to allow sequencing if necessary. At the refinement level, we are able to model sequencing. that is, if several operations are to be called call sequencing. So, we are able to call state change operations (i.e., *Train_TransFarNear*) in sequence with operations for actions from the included machine *Train* if there are some modelled.

- The *Types* machine models shared types and data. This separation of concerns provides a clear way of identifying the publicly visible information and allows all components of the system to use the same definitions.

Note that the actual translation from UML diagrams into B is not of interest for the study undertaken in this paper. Of interest are the formal concepts coming with the B language and whether and how it is applicable in a joint development process. Interested readers can find proposals on UML to B transformation in [MS99,LP01,LS02,SBO03,ML02].



```
MACHINE Types
SETS
OBJECTS;
TRAIN_STATES =
      {far, near, on}
CONSTANTS
TRAIN
PROPERTIES
TRAIN ⊆ OBJECTS
END
```

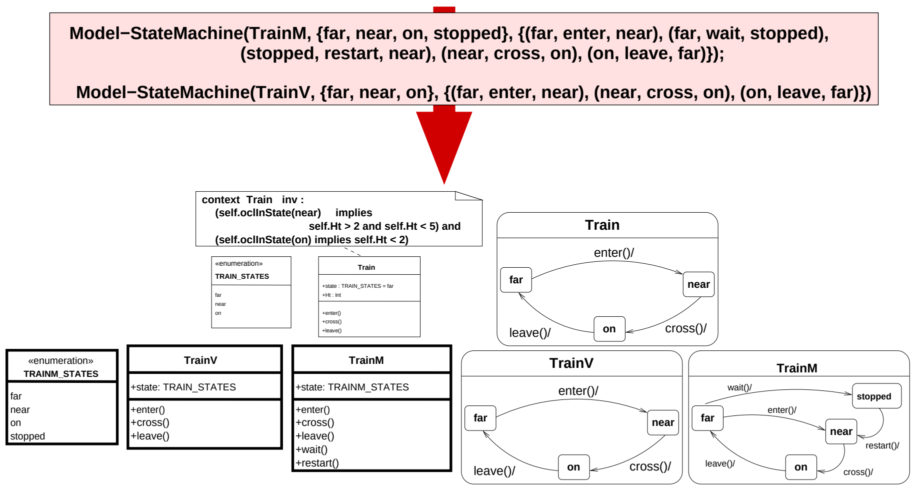
```
MACHINE Train
SEES Types
VARIABLES
train, Train_state
INVARIANT
train ⊆ TRAIN ∧
Train_state ∈
          train → TRAIN_STATES

INITIALISATION
train := ∅ ||
Train_state := ∅
OPERATIONS
Train_TransFarNear(oo) =
PRE oo ∈ train ∧
      Train_state(oo) = far
THEN
      Train_state(oo) := near
END;
...
END
```

```
MACHINE System
SEES Types
INCLUDES aT.Train
OPERATIONS
enter(oo) =
PRE oo ∈ OBJECTS
THEN
      IF oo ∈ aT.train
      THEN
            IF aT.Train_state(oo) = far
            THEN
              aT.Train_TransFarNear(oo)
            END
            ELSE skip END
END;
...
END
```

```
REFINEMENT System_ref
REFINES System
SEES Types
INCLUDES aT.Train
OPERATIONS
enter(oo) =
PRE oo ∈ OBJECTS
THEN
      IF oo ∈ aT.train
      THEN
            IF aT.Train_state(oo) = far
            THEN
              aT.Train_TransFarNear(oo)
            END
            ELSE skip END
END;
...
END
```

Fig. 3 Application of the Model-StateMachine operator to initialise the development

3.2 Second development step : introduction of different kinds of train


Model−StateMachine(TrainM, {far, near, on, stopped}, {(far, enter, near), (far, wait, stopped), (stopped, restart, near), (near, cross, on), (on, leave, far)});
Model−StateMachine(TrainV, {far, near, on}, {(far, enter, near), (near, cross, on), (on, leave, far)})
context Train inv : (self.oclInState(near) implies self.Ht > 2 and self.Ht < 5) and (self.oclInState(on) implies self.Ht < 2)
«enumeration» TRAIN_STATES
far
near
on
Train
+state : TRAIN_STATES = far
+Ht : Int
+enter()
+cross()
+leave()
Train
enter()/
far
near
on
leave()/
cross()/
«enumeration» TRAINM_STATES
far
near
on
stopped
TrainV
+state: TRAIN_STATES
+enter()
+cross()
+leave()
TrainM
+state: TRAINM_STATES
+enter()
+cross()
+leave()
+wait()
+restart()
TrainV
enter()/
far
near
on
leave()/
cross()/
TrainM
wait()/
stopped
far
enter()/
near
restart()/
leave()/
on
cross()/

```
MACHINE Types
SETS
...
TRAINM_STATES
    {far, near, on, stopped}

CONSTANTS
...
    TRAINV, TRAINM
PROPERTIES
...
    TRAINM ⊆ OBJECTS ∧
    TRAINV ⊆ OBJECTS
END
```

```
MACHINE Train
...
END
```

```
MACHINE System
SEES Types
INCLUDES aT.Train,
    aTM.TrainM, aTV.TrainV
OPERATIONS
enter(oo) = ...
...
    wait(oo) = ...
...
END
```

```
MACHINE TrainV
SEES Types
VARIABLES
trainv, TrainV_state

INVARIANT
trainv ⊆ TRAINV ∧
TrainV_state ∈
    trainv → TRAIN_STATES
INITIALISATION
...
OPERATIONS
TrainV_TransFarNear(oo) = ...
...
END
```

```
MACHINE TrainM
SEES Types
VARIABLES
trainm, TrainM_state

INVARIANT
trainm ⊆ TRAINM ∧
TrainM_state ∈
    trainm → TRAINM_STATES

INITIALISATION
...
OPERATIONS
TrainM_TransFarNear(oo) = ...
TrainM_TransFarStopped(oo) =
...
...
END
```

```
REFINEMENT System_ref
REFINES System
SEES Types
INCLUDES aT.Train,
    aTM.TrainM, aTV.TrainV
OPERATIONS

    enter(oo) =
    PRE oo ∈ OBJECTS
    THEN
    IF oo ∈ aT.train
    THEN
        IF aT.Train_state(oo) = far
        THEN
        aT.Train_TransFarNear(oo)
        END
        ELSE IF oo ∈ aTV.trainv
        THEN
        aTV.TranV_TransFarNear(oo)

        ELSE IF oo ∈ aTM.trainm

        THEN
        aTM.TranM_TransFarNear(oo)

        ELSE skip END
    END
    END
    END;

    wait(oo) = ...
    restart(oo) = ...
...
END
```

Fig. 4 Introduction of two kinds of trains

A further analysis of the problem indicates that different kinds of trains are authorised to travel on the GRC: freight trains and passenger trains. The following characteristics are identified:

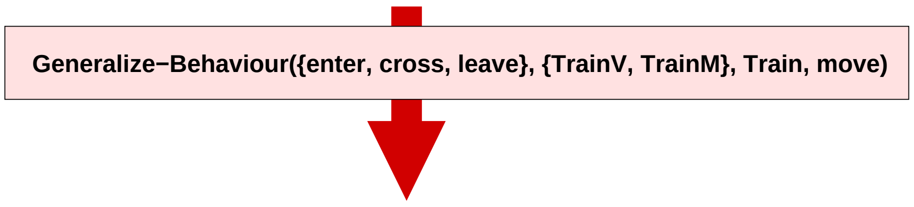
- freight trains can stop when they reach the state *far* after the event *wait* occurs. They go from the state *stopped* to the state *near* when the event *restart* occurs.

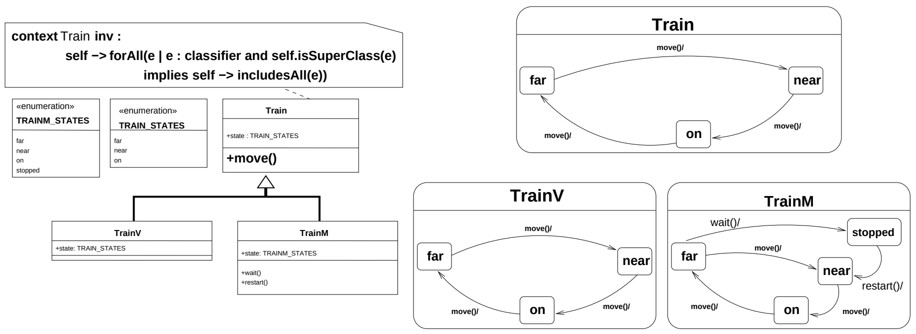
To introduce the different trains, we can choose between at least two development approaches: passenger and freight trains can be modelled independently from the *Train* entity, or they can be modelled as specialisation of the *Train* entity. Let us apply the first approach which corresponds to a bottom-up strategy.
We use again the *Model-StateMachine* operator, once for the freight trains (*TrainM*) and once for the passenger train (*TrainV*).
The new specification state is presented in Fig. 4. Two classes, one enumerated type and two state diagrams have been introduced in the UML view. Two machines have been introduced and three other entities (*Types*, *System*, and *System_ref*) have been updated in the B view.

3.3 *Third development step : Generalisation*

The previous development steps have produced three unconnected entities. A close look on the diagrams and machines reveals strong similarities. In fact, we have modelled twice the same general behaviour. Moreover, we have now enough knowledge of the problem to realize that *enter*, *cross* and *leave* are three instances of the same behaviour: *move*. This situation is quite common while developing specification and can be solved by generalising. A generalisation operator,
`Generalize-Behaviour`, models this approach. We select the parameters to indicate that *TrainV* and *TrainM* are sub-kinds of *Train* and that one operation, *move*, replaces the other three.

The new specification state is presented Fig. 5. We can note that the UML view has been augmented with inheritance relations and the definition of operations *enter*, *cross* and *leave* have been removed from the subclasses. They are also now modelled by the generic operation, *move*, in the superclass. It is implicitly inherited by the subclasses. The B view shows modifications in the corresponding machines.


Generalize−Behaviour({enter, cross, leave}, {TrainV, TrainM}, Train, move)



```
MACHINE Types
SETS
...
PROPERTIES
    TRAINV ⊆ TRAIN ∧
    TRAINM ⊆ TRAIN
END
```

```
MACHINE Train
...
OPERATIONS
Train_TransFarNear(oo) = ...
Train_TransNearOn(oo) = ...
Train_TransOnFar(oo) = ...
END
```

```
MACHINE TrainV
SEES Types
    EXTENDS Train
...
INVARIANT
...
    trainv ⊆ train
...
OPERATIONS
TrainV_TransFarNear(oo) =
...
END
```

```
MACHINE TrainM
SEES Types
    EXTENDS Train
...
INVARIANT
...
TrainM_Ht ∈ trainm → NAT
    trainm ⊆ train
...
OPERATIONS
TrainM_TransFarNear(oo) =
...
END
```

```
MACHINE System
SEES Types
INCLUDES aT.Train, aTM.TrainM
         aTV.TrainV
OPERATIONS
    move(oo) = ...
...
END
```

```
REFINEMENT System_ref
REFINES System
SEES Types
INCLUDES aT.Train, aTM.TrainM
         aTV.TrainV

OPERATIONS
    move(oo) =
    PRE oo ∈ OBJECTS
    THEN
    IF oo ∈ aT.train
    THEN
        IF aT.Train_state(oo) = far
        THEN
        aT.Train_TransFarNear(oo)
        END
        ELSE IF aT.Train_state(oo) =
        near
        THEN
        aT.Train_TransNearOn(oo)
        ELSE IF aT.Train_state(oo) =
        on
        THEN
        aT.Train_TransOnFar(oo)
        ELSE skip END
    END
    END ||
    IF oo ∈ aTV.trainv
        ... ||
    IF oo ∈ aTM.trainm
    ...
    END;

...
END
```

Fig. 5 Application of the Generalize-Behaviour operator on Fig 4

4 Operator's correctness

We derive the correctness of an operator from the correctness of specifications that it produces. This is done by verifying the four conditions of the consistency relation. To illustrate this, we take the specification of Fig 3 obtained by the application of *Model-StateMachine* operator.
Let : $\langle SpecUML', \; SpecB' \rangle$ the UML and B specification of Fig 3

1 **Syntactic conformance.** Both specifications must be checked for syntax and type correctness with their corresponding support tool. The B support tool we use for this case study, *b4free* [B4F], confirms the well-formedness of the text shown in Fig. 3. The UML diagrams are well-formed according to *ArgoUML*, the UML tool we have used.

2 **Local consistency.** The definition of operator correctness uses the strong hypothesis that each view in the initial state is internally consistent. While this condition is not much more than the well-formedness for the UML, it means full logical consistency for the B part. The checking of *SpecB'* follows the usual approach of the B method: to check initialisation, to check pre and postconditions of operations with respect to the preservation of machine invariants, and to check inter-machine relations such as sees, includes or refinements. We have submitted *SpecB'* to the b4free tool.

```
Project status
+-----------+----+-----+-----+-----+-----+-----+
| COMPONENT | TC | POG | Obv | nPO | nUn | %Pr |
+-----------+----+-----+-----+-----+-----+-----+
| System    | OK | OK  |  10 |   0 |   0 | 100 |
| System_ref| OK | OK  |  22 |   0 |   0 | 100 |
| Train     | OK | OK  |   7 |   8 |   0 | 100 |
| Types     | OK | OK  |   1 |   0 |   0 | 100 |
+-----------+----+-----+-----+-----+-----+-----+
| TOTAL     | OK | OK  |  40 |   8 |   0 | 100 |
+-----------+----+-----+-----+-----+-----+-----+
```

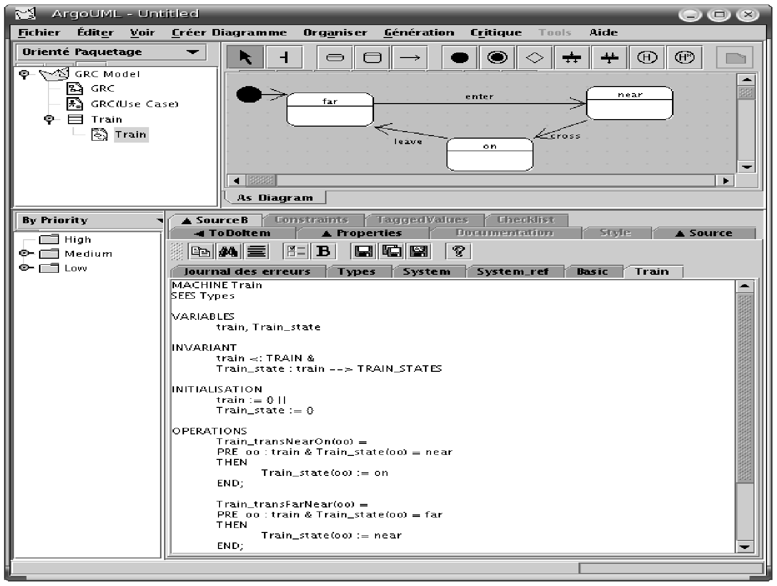
Fig. 6 Result of the verification of the B specification

All proof obligations generated by the tool have been discharged. Fig. 6 shows the summary of the verification printed by the tool. This gives us a first feedback on the internal consistency of *SpecB'*.

3 **Elements traceability** is proved by verifying that
$ID(\mathbb{T}_{U \to B}(SpecUML')) = ID(SpecB')$. All new names introduced by the operator are present.

4 **Semantics preservation.** Our strategy to verify this condition is to submit the B specification of Fig. 7 to the *b4free* tool and compare the proof results with those obtained previously for the B specification of Fig 3. So, due to the elements traceability condition, we conclude that *SpecB'* satisfies the same requirement than it UML counterpart. This has been checked true on our example.

The four conditions of the consistency relation hold for the first development state, we can assume that the *Model-StateMachine* operator works correctly.


ArgoUML - Untitled
Fichier Éditer Voir Créer Diagramme Organiser Génération Critique Tools Aide
Orienté Paquetage
GRC Model
GRC
GRC(Use Case)
Train
Train
far
enter
near
leave
on
cross
As Diagram
By Priority
High
Medium
Low
SourceB Constraints TaggedValues Checklist
ToDoItem Properties Documentation Style Source
Journal des erreurs Types System System_ref Basic Train

```
MACHINE Train
SEES Types

VARIABLES
        train, Train_state

INVARIANT
        train <: TRAIN &
        Train_state : train --> TRAIN_STATES

INITIALISATION
        train := ∅ ||
        Train_state := ∅

OPERATIONS
        Train_transNearOn(oo) =
        PRE  oo : train & Train_state(oo) = near
        THEN
                Train_state(oo) := on
        END;

        Train_transFarNear(oo) =
        PRE  oo : train & Train_state(oo) = far
        THEN
                Train_state(oo) := near
        END;
```

Fig. 7 Specification of the Train obtained by systematic UML to B transformation of Fig 3 with ArgoUML+B

5 Conclusion

This paper presents a specification development process which integrates the use of several formalisms. The key notion is the operator which models a development strategy while ensuring that the multi-view specification evolves consistently. The idea to mix different formalisms is not new but was hampered by the problem of maintaining the consistency between the two specifications. Operators solve this problem. They enable users to develop specifications in an intuitive fashion by separating the use of design concepts from the technical details of how they are captured in specification languages. They offer flexibility since it is possible to define libraries of operators capturing alternative definitions of particular concepts and strategies. They allow us to model the development of a specification as a process of successive approximation process. The purpose of operators is to capture the specifiers' knowledge.

The benefits of the approach can be summed up as follows:

- separation of concern. Operators enable the specifier on methodological issues and on problem solving issues rather than to focus on how to express them in the target languages;
- documentation. The use of two complementary languages, one graphical and object-oriented and the other formal, makes the specification easier to understand and help the developers to verify and refine the system under development;
- support for guidance. At any stage of the construction process, the specifier knows what remains to be done. Libraries of operators with a liberal use of the “remain to be done” clause can be constructed to model and enforce particular

development strategies. In addition, operators preconditions lower the risk of mis-using operators;

- correctness by construction. As the correctness of each operator has been defined, the specification obtained by the application of operators is proved to be correct.

Operators can be compared with specification templates introduced in [Tur96], where a template formalises a Lotos specification style for OSI as a fragment of specification text that can be conveniently retrieved and inserted in a specification. To enhance the value of such templates and to increase their generality, templates are parameterised.

References

[Abr96] J.R. Abrial. *The B Book -Assigning Programs to Meanings.-*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[B4F] B4Free. avaible at : http://www.b4free.com/.

[BCL96] Oxford(UK) B-Core(UK) Ltd. *B-Toolkit User's Manual*. 1996.

[CD94] S. Cook and J. Daniels. Let's get formal. *Journal of Object-Oriented Programming (JOOP)*, pages 22–24 and 64–66, 1994.

[CDD+90] D.A. Carrington, D. Duke, R. Duke, P. King, G.A. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In *Formal Description Techniques II, FORTES'89*, pages 281–296, 1990.

[HM04] F. Houda and S. Merz. Transformation de spécifications B en diagrammes UML. In *Proceedings of AFADL'04*, 2004.

[IL04] A. Idani and Y. Ledru. Object Oriented Concepts Identification from Formal B Specifications. In *9th Int.Workshop on Formal Methods for Industrial Critical Systems, FMICS'04*, 2004.

[JS00] L. Jansen and E. Schneider. Traffic control system case study: Problem description and a note on domain-based software specification. Technical report, Colorado State University, January, 2000.

[Lan91] K. Lano. Z++, An Object-orientated Extension to Z. In *Proceedings of the Fifth Annual Z User Meeting*, pages 151–172. Springer-Verlag, 1991.

[LP01] R. Laleau and F. Polack. A Rigorous Metamodel for UML Static Conceptual Modelling of Information Systems. In *Advanced Information Systems Engineering. 13th Int. Conf., CAiSE 2001*, volume 2068 of *LNCS*, pages 402–416. Springer, 2001.

[LP02] R. Laleau and F. Polack. Coming and Going from UML to B : A Proposal to support Traceability in Rigorous IS Development. In *ZB'2002 – Formal Specification and Development in Z and B*, pages 517–534, 2002.

[LS01] H. Ledang and J. Souquières. Modeling class operations in B: application to UML behavioral diagrams. *ASE2001: 16th IEEE Int. Conf. on Automated Software Engineering, IEEE Computer Society*, 2001.

[LS02] H. Ledang and J. Souquières. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. In *APSEC 2002, IEEE Computer Society*, 2002.

[LSC03] H. Ledang, J. Souquières, and S. Charles. ArgoUML+B : Un outil de transformation systématique de spécifications UML vers B. In *Proceedings of AFADL'03*, 2003.

[ML02] R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. In L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, editors, *Workshop on Consistency Problems in UML-based Software Development*, pages 91–105, 2002.

[MS99] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *Proceedings of the Formal Method Conference*, number 1708 in LNCS, pages 875—895. Springer-Verlag, 1999.

[OSJ05] D. Okalas Ossami, J. Souquières, and J-P. Jacquot. Consistency in UML and B multi-view specifications. In LNCS, editor, *Proc. of the Int. Conf. on Integrated Formal Methods, IFM'05*, number 3771, pages 386–405, 2005.

[RJB97] J. Rumbaugh, I. Jacobsen, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. ISBN 0-201-30998-X.

[SB] C. Snook and M. Buttler. U2B: a tool for combining UML and B. Avaible at http://www.ecs.soton.ac.uk/ cfs/U2Bdownloads/.

[SBO03] C. Snook, M. Butler, and I. Oliver. Towards a UML profile for UML-B. Technical report, DSSE-TR-2003-3, Electronics and Computer Science, University of Southampton, 2003.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

[STE98] STERIA. *Manuel de référence du langage B*. -ClearSy-, novembre, 1998.

[Tur96] K. J. Turner. Relating architecture and specification. *Computer Networks and ISDN Systems*, April 1996.

[TV01] B. Tatibouet and J. C. Voisinet. jBtools and B2UML : a plateform and a tool to provide a UML class diagram since a B specification. In *ICSSEA : 14th Int. Conf. on Software and Systems Engineering and Their Applications*, 2001.

[TV03] B. Tatibouet and J.-C. Voisinet. Generating statecharts from B specifications. In *16th Int. Conf. Software & Systems Engineering and their applications, ICSSEA'2003*, 2003.

6 Appendix

Operator Model-StateMachine
Description. This operator models a state dependent behaviour of an entity.
Parameters

- $EntityName : Names$
- $states\_set : \mathbb{F}(names)$
- $transitions\_set : \mathbb{F}(TRANS)$

Application conditions

General to SpecUML and SpecB

- $\forall s_i.(s_i \in states\_set \Rightarrow s_i \notin IDUsedIn(EntityName))$

Definition

```
IF $EntityName \notin MachinesID(SpecB) \cup RefinementsID(SpecB)$
THEN
    $Introduce\text{-}Entity(EntityName)$ ;
END ;

IF $StateMachineOf(EntityName) \notin Spec\_StateMachines(SpecUML)$
THEN
```

$\mathcal{O}_B$	$\mathcal{O}_{UML}$
$skip$	$AddStateMachine(EntityName)$

;

```
    IF $\{''System'', ``System\_ref''\} \cap (MachinesID(SpecB) \cup RefinementsID(SpecB)) = \varnothing$
    THEN
```

$\mathcal{O}_B$	$\mathcal{O}_{UML}$
$AddSystemMachines(\{EntityName\})$	$skip$

```
    END
END ;

IF $StateMachineOf(EntityName) \in Spec\_StateMachines(SpecUML)$
THEN
    IF $StatesTypeID(EntityName) \notin TypesID(Spec)$
    THEN
        $Introduce-Enumerated-Data-Type(generateStatesType(EntityName), states\_set)$
    END ;

    IF $StatesTypeID(EntityName) \in TypesID(Spec) \wedge states\_set - StatesOf(EntityName) \neq \varnothing$
    THEN
```

$\mathcal{O}_B$	$\mathcal{O}_{UML}$
$AddSetElements(``Types''$ $StatesTypeID(EntityName),$ $states\_set - StatesOf(EntityName))$	$AddTypeElements($ $StatesTypeID(EntityName),$ $states\_set - StatesOf(EntityName))$

```
    END ;
```

$\mathcal{O}_B$	$\mathcal{O}_{UML}$
$skip$	$AddStates(StateMachineOf(EntityName), states\_set)$

```
END ;

FOR $t$ IN $transitions\_set - TransitionsOf(StateMachineOf(EntityName))$
DO
    $Introduce-Transition(StateMachineOf(EntityName), t)$
END
```

Fig. 8 Formal description of Model-StateMachine