From Event-B Specifications to Programs for Distributed Algorithms

Mohamed Tounsi
Univ. Bordeaux, LaBRI
F-33405 Talence,France
tounsi@labri.fr

Mohamed Mosbah
Univ. Bordeaux, LaBRI
F-33405 Talence, France
mosbah@labri.fr

Dominique Méry
Univ. Lorraine, LORIA
F-54506 Vanœuvre-ls-Nancy, France
mery@loria.fr

***Abstract*—Formal proofs of distributed algorithms are long, hard and tedious. We propose a general approach, based on the formal method Event-B, to automatically generate correct programs of distributed algorithms. Our approach is implemented with a translation tool, called B2Visidia, that generates Java code from an Event-B specification related to distributed algorithms. The resulting code can be run on classical distributed computing systems. To execute the induced programs, we use a tool called Visidia that can be used for experimenting, testing and visualizing programs of distributed algorithms.**

***Index Terms*—Formal methods, Event-B, Distributed algorithms, Local computations, Visidia, Distributed systems**

I. Introduction

A. Overview

Distributed systems [1], [2] have gained much economical importance due to an increasing number of web based applications and enterprise wide cooperating software. Moreover, the technological advances of computers and networks combined with middle-ware solutions help to avoid physical hardware problems such as low speed, heterogeneity, failure, etc. Thus, distributed systems have become an economical challenge for many companies. Examples of distributed applications include manufacturing, banking, process control, or weather forecast computations.

However, the development of distributed systems is yet not well understood. The design, validation, verification and debugging remain a hard task for most programmers and computer scientists. This is due to the intrinsic complexity of distributed algorithms. Correctness of these algorithms is crucial because it gives us confidence that distributed systems perform as designed and do not behave harmfully. Unfortunately, this imposes a heavy burden to the designer during the development process. Since, these algorithms are characterized by the lack of knowledge of the global state and the non determinism of the execution of the processes.

Formal methods provide a real help for expressing correctness with respect to safety properties in the design of distributed algorithms. Particularly, *correct-by-construction* is a well suited approach which provides an easy way to prove algorithms. Its main idea relies upon a formal development following a top/down approach, which is clearly well known in earlier works of Dijkstra [3], and to use refinement for ensuring the correctness of the resulting algorithms. This incremental proof-based process allows to simplify the proofs and to validate integration of requirements. Event-B [4] modeling language is supporting this methodological proposal suggesting proof-based guidelines. Event-B is supported by a tool called RODIN [5].

A distributed system is modeled by a simple, connected and undirected graph where nodes denote processors, and edges denote communication links. Distributed algorithms are considered with respect to the local computations models [6]: nodes exchange their states and accordingly to those, they do a computation step. More precisely, a distributed algorithm is simply given by a set of rules. A run of the algorithm consists in applying the relabelling rules specified by the algorithm until no rule is applicable, which terminates the execution. The relabelling rules are applied asynchronously and nondeterministically, which means that given the initial labelling usually many different runs are possible. The distributed aspect means that two consecutive non-overlapping steps may be applied in any order and in particular in parallel.

B. Contribution

In this work, we propose a new approach to automatically implement Event-B models of distributed algorithms. More precisely, we develop a general and unified method and a tool called B2Visidia to generate a Java implementation of distributed algorithm specified in Event-B. The generated code can be run on any distributed computing system that has a Java interface to manage message exchanges and state node modifications. In this paper, we use the Visidia tool [7] which offers a high level interface to implement programs of distributed algorithms.

The translation process of B2Visidia consists of transforming a concrete Event-B model into a semantically equivalent Java code. However, a model is translatable by B2Visidia, only if it contains deterministic substitutions and uses a subset of Event-B language, called B2Visidia language. The purpose of this language is to reduce the used syntax, to guarantee a non-ambiguous translation and to allow specification of any local computation model. This language is based on our formal framework which combines local computation models and refinement to prove the correctness of many classes of distributed algorithms [8].

978-0-7695-5002-2/13 $26.00 © 2013 IEEE
DOI 10.1109/WETICE.2013.44

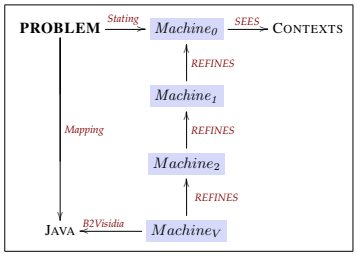


Fig. 1. B2Visidia position in an Event-B development

The following diagram (see Fig.1), shows the B2Visidia position in an Event-B development and explains how we can generate from the problem analysis a Visidia solution.

- Contexts states properties of graphs and the mathematical problem to solve (election, spanning tree, ...)
- A model $Machine_0$ expresses the specification of the problem to solve by events stating a relation between the initial states and the final states.
- The refinement of $Machine_0$ into a model $Machine_1$. Now $Machine_1$ can express the inductive property which allows to express the computation in the local computation model.
- The next refinement of $Machine_1$ (called $Machine_2$) allows producing a set of events corresponding to the set of relabelling rules.
- The next refinement of $Machine_2$ (called $Machine_V$) is a refinement for producing an Event-B model which respect to the B2Visidia language.
- A *Java* code is generated from $Machine_V$ by using the B2Visidia approach.

We think that, the most important idea behind the construction of B2Visidia tool is to help the designer for proving specifications of distributed algorithms. Also, B2Visidia can be considered as an academic tool that assists students in their studies. Furthermore, it aims at providing a library of proved algorithms under Visidia.

C. State of art

The most representative works dealing with code generation are those of Wright S. [9], Mery D. et al. [10] and Edmunds A. et al. [11]. The first one proposes an approach to generate a C code from an Event-B specification. The approach can only support a small sub-set of Event-B language and it is implemented as a plug-in to the RODIN platform. The second work is considered as an evolution of the first one. In this work, authors proposed a tool to generate up to four programming languages : C, C++, C#, and Java. The translation tool is rigorously developed with safety properties preservation. The third one presents an approach for generating code, for concurrent programs, from Event-B specifications. B2Visidia is different from these works since it is intended to distributed algorithms. Also, it generates a Java implementation which can be only executed under Visidia.

D. Organization of the paper

The paper is organized as follows: Section 2 presents a B2Visidia overview. Section 3 describes the B2Visidia architecture. Section 4 presents basic concepts of the Event-B modeling language and gives an example of a distributed algorithm developed with Event-B. Section 5 introduces the B2Visidia language and explains how a Java code can be generated from an Event-B specification. Finally, we conclude this paper by summarizing our idea and by describing future directions.

II. B2Visidia Overview

The purpose of this section is to show how B2Visidia can be introduced within a development process of distributed algorithms. As presented in Fig.2, user (the designer) is the main actor in this process. Firstly, he provides an Event-B specification of a distributed algorithm. To do this, he can use either the RODIN platform, or the B2Visidia tool. However, RODIN guarantees correction of the syntax and the proof obligations in contrast to B2Visidia which ensures only the syntax correctness.

Secondly, the Event-B specification is translated. Therefore, if user chooses RODIN in the first step, B2Visiadia performs a filtering and a rewriting operations on the RODIN source file to be ready for the translation. Because, as we show in Section V, many sections in a specification are not useful for the translation.

Thirdly, once the Java code is generated, the user have to launch Visidia, execute the code and test the algorithm specification. Here, two possibilities are conceivable :

- the test is successful, user can then conclude that the algorithm specification is correct.
- otherwise, user locates the potential problem, corrects and repeats the process at the outset.

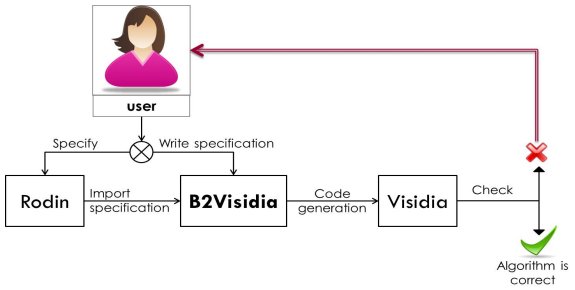


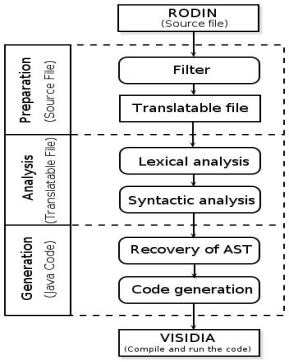
Fig. 2. Development process of distributed algorithms

III. B2Visidia architecture

Translating an Event-B model into a concrete language (such as Java language) is not possible in one shot. Since, Event-B specifies systems without taking into account their implementations. Therefore, it keeps specifications in a high level of abstraction and usually, it omits some implementation details (the synchronization type and the communication

paradigm for instance). Also, a specification can include global conditions which complicates the extraction of the local data of each processor in the network. The other difficulty can rely on the extraction of useful variables, and/or their types, for the implementation; because a specification may include variables which are only defined to be used in the proof processes.

B2Visidia makes it possible to translate an Event-B specification, containing useful annotations, into a Java code for Visidia. Our approach includes three steps: the initial step consists in preparing the source file (XML file) stored in RODIN platform. The goal of this step is to generate a simple and a translatable version.



To this end, we use Tom [12] which is a language and a software environment suitable for programming various transformations on trees/terms. Also, it can be used to match and rewrite XML documents. Once the file rewritten, we perform, in the second step a lexical and syntactic analysis of the translatable file to build an Abstract Syntax Tree (AST) of the algorithm specification. We use Java Cup[1] for the syntactic analysis and the scanner generator Jflex[2] for the lexical analysis. In the final step, we translate the AST nodes and generate the corresponding Java code for Visidia.

IV. Event-B language

The Event-B modeling language allows to define mathematical structures into contexts and formal model of a system into machines. The modeling process starts by identifying the domain of the problem expressed by means of context. This latter consists of the following elements: a name, a list of distinct carrier sets, a list of distinct constants and a list of named properties. The machine describes a system characterized by a finite list of events modifying a state variable; an operational interpretation of an Event-B machine states that traces of the current model can be generated from the initial states by applying events. A machine may encapsulate a set of mathematical items, variables, invariants and a set of events on these variables. An invariant is defined to be a predicate that holds in all reachable states. An event is decomposed into a guard that specifies under which circumstances it might occur and some generalized substitutions called actions that define the state transition associated with the event.

A context associated with a given machine defines the way this machine is parameterized and can thus be instantiated [13]. A machine M may see a context C, this means that all carrier sets and constants defined in C can be used in M. A machine can be built and asserted to be a refinement of another machine [14]. Consequently, the new machine is named a refinement or a concrete version of the first machine. The refinement of a machine enriches a model in a step-by-step approach. It provides a way to strengthen invariants and to add details to a model. This is done by extending the list of state variables, by refining each abstract event into a corresponding concrete version and by adding new events. Likewise, a context can be extended to another context.

However, an Event-B specification is considered as correct only if each machine, as well as the process of refinement, are proved by adequate theorems named Proof Obligations (PO); i.e events preserve the invariant(s) and each event is feasible. The management of proof obligations is a technical task supported by RODIN tool [5], which provides an environment for developing *correct-by-construction* models for software-based systems.

A. Event-B specification of a spanning tree algorithm

We present in this section an example of a distributed algorithm encoded by local computations model. The chosen algorithm computes a spanning tree of a network. As this algorithm was presented in [15], we only show some parts from its Event-B specification. Essentially, parts that are used in the next section for illustrating the translation process.

1) Algorithm presentation: Assume that a unique given processor is in an "active" state (encoded by the label **A**), all other processors being in some "neutral" state (label **N**) and that all links are in some "passive" state (label **F**ALSE). The tree initially contains the unique active node. At any step of the computation, an active node may activate one of its neutral neighbours and mark the corresponding link which gets the new label **T**RUE. This computation stops as soon as all the processors have been activated. The spanning tree is then obtained by considering all the links with label **T**RUE. An elementary step in this computation may be depicted as a *relabelling step* by means of the following relabelling rule R which describes the corresponding label modifications.

$$\text{R: } \overset{A}{\bullet} \overset{FALSE}{\text{———}} \overset{N}{\bullet} \longrightarrow \overset{A}{\bullet} \overset{TRUE}{\text{———}} \overset{A}{\bullet}$$

An application of this relabelling rule on a given graph consists in (i) finding in the graph a subgraph isomorphic to the left-hand-side of the rule and (ii) modifying its labels according to the right-hand-side of the rule.

2) Formal specification: We begin by defining, in the Event-B context, the network on which the algorithm operates. Then, we specify the set of all possible spanning trees on this network. Formally, a network can be straightforwardly modeled as a connected, non-oriented and simple graph where nodes *ND* denote processors and edges *g* denote direct communication links. We define *trees* as the set of spanning trees in *g* [15]. *r* is a root node. $trees = \{t \in ND \setminus \{r\} \rightarrow ND \wedge t \subseteq g \wedge (\forall q \cdot q \subseteq ND \wedge r \in q \wedge t^{-1}[q] \subseteq q \Rightarrow ND = q)\}$

The first level $Machine_0$, expresses only the goal of the distributed algorithm and does not describe how the solution is computed.

[1]http://www2.cs.tum.edu/projects/cup
[2]http://jflex.de

oneshot
any
tr
where
$grd1 : tr = \varnothing$
then
$act1 : tr :\in trees$
end

This goal is stated by a "oneshot" event [8] which returns a spanning tree among the non-empty set of possible spanning trees $trees$ in g. tr $\in$ ND $\leftrightarrow$ ND.

The second level $Machine_1$ remains in a high level abstraction, it encodes the algorithms and computes its result without considering relabeling rules. It introduces a new event *Progress* which gradually computes the spanning tree. New variables are added: tr_nodes is the set of nodes which belong to tree under construction new_tree. The variable $remaining\_nodes$ is the set of nodes which do not belong to new_tree.

Progress
any
$s1, s2$
where
$grd1 : s1 \in tr\_nodes \wedge s2 \in remaining\_nodes$
$grd2 : s1 \mapsto s2 \in g$
then
$act1 : new\_tree := new\_tree \cup \{s1 \mapsto s2\}$
$act2 : tr\_nodes := tr\_nodes \cup \{s2\}$
$act3 : remaining\_nodes := remaining\_nodes \setminus \{s2\}$
end

The third level $Machine_2$ introduces labels of nodes and edges. Now, the machine can specify the local label modification and encode the relabeling rule described above. In this level, the oneshot event still exists but it is more concrete. Let lab and $Mark$ be two variables which describe respectively the node and the edge states : $lab \in ND \rightarrow node\_label$ and $Mark \in g \rightarrow edge\_label$ where $node\_label$ and $edge\_label$ are defined in the context as the set of possible labels for respectively nodes and edges.

RelabelingRule
any
$s1, s2$
where
$grd1 : s1 \mapsto s2 \in g$
$grd2 : lab(s1) = A \wedge lab(s2) = N$
then
$act1 : lab(s2) := A$
$act2 : Mark(s1 \mapsto s2) := \text{TRUE}$
end

These variables replace the abstract ones as follows: $tr\_node = lab^{-1}[\{A\}]$, $remaining\_nodes = lab^{-1}[\{N\}]$ and $new\_tree = Mark^{-1}[\{\text{TRUE}\}]$.

V. How translate an Event-B specification ?

The purpose of this section is to show how B2Visidia can translate an Event-B specification of a distributed algorithm. First, we present briefly the B2Visidia language which is a translatable sub-language of Event-B. It serves to specify the $Machine_V$ and defines conditions that must be respected in order to ensure a correct translation of the machine. After that, we give an idea about the translation step. The purpose of this section is not to provide all translation rules but rather to show the feasibility of such a translation.

A. B2Visidia Language

The B2Visidia language is considered as a translatable subset of the Event-B language. We kept the same definition of the associativity and priorities of operators as it has been adopted by Event-B. Its syntax is simplified to guarantee a non-ambiguous translation. Also, it can be used to specify any local computations model. It relies upon our earlier work [8] which proposes a framework combining local computations models and refinement to prove the correctness of a large class of distributed algorithms.

1) Context specification: In our approach, only machines are translated. Since, some context informations can be retrieved in the machines. In addition and in order to avoid parsing of the context, we have imposed some conventional annotations : ND to encode nodes, g to encode edges, ID to encode a distinct identifier number of a node in the graph and $card$ to encode the node degree (number of its neighbors).

2) Machine specification: A machine may contain sections (theorems, variant, refines, uses.. for instance) which serve to prove the correctness of the design, but they are not appropriate for expressing the functional aspect of the distributed algorithms. These sections do not capture the behavior of the distributed algorithm and so they will be eliminated once the source file is filtered. We can point out that a translatable machine should contain the following sections: a machine name, a synchronization type, a set of variables, invariants and events.

a) Synchronization types: In order to implement a local computations model, it is necessary to know the synchronization type which it uses. This information cannot be introduced as a parameter in the Rodin platform because it is considered as an implementation detail. To bridge this gap, we use Rodin annotation to provide a further guidance for the B2Visidia tool and then it will be an integral part of the defined language. We define an annotation as a comment added to the specification to guide the translation process. It is differentiated from other comments by the "#" character. Synchronization annotation is added just after the machine name. Synchronization annotations are of three different types: #LC0, #LC1 and #LC2. In a LC0 computation step, only labels attached to two connected nodes may be modified. In a LC1 computation step, the label attached to the center of a star is modified according to some rules depending on the labels of the star, labels of the leaves are not modified. In a LC2 computation step, labels attached to the center and to the leaves of a star may be modified according to some rules depending on the labels of the star.

Due to the lack of space and as the example developed above uses the LC0 synchronization, the remainder of the section will be developed with respect to LC0 synchronization.

b) Variables: In the variables component, each label is specified by a variable that we called *"visidia_variable"*. In the example presented in Section IV-A2, *lab* and *Mark* are two *"visidia_variable"*. The first one encodes the node state and the second one the edge state.

c) Invariants: "visidia_variable" are declared in the invariant component according to the following syntax[3]:

<visidia_invariant> ::= <visidia_variable> '∈' <label_type>

[3]We use the Extended Backus-Naur Form (EBNF) to describe syntax. In that notation, non-terminals are surrounded by angle brackets and terminals are surrounded by single quotes.

<label_type> ::= 'ND' <relational_set_op> <set_expression> # <type>
| 'g' <relational_set_op> <set_expression>

<label_type> specifies the *"visidia_variable"* type. Its first [resp. second] form serves to define <visidia_variable> that encodes the node [resp. edge] state. Also, it reveals the type of the label by means of a specific annotation ("$\#int$" for integral label for instance). *<relational_set_op>* is the set of operators that are used to build sets of relations or functions (total function '$\rightarrow$' for example). Since, $type$ annotation can reveal the type of the variable, *<set_expression>* will not be analyzed by B2Visidia. For our example, "$lab \in ND \rightarrow node\_label\#string$" defines a variable which encodes a node label having "*String*" as type. However, Visidia tool has some weakness, e.g., it allows only one action on the edge state (the marking). This means that, an edge can have only two possible states : Marked or not Marked. In that case, the type of a "$visidia\_variable$" can be only $boolean$ where "$true$" decrypts the state of a marked edge and "$false$" decodes the state of a non-marked edge.

d) Events : An event consists of three elements: (1) a name, (2) a list of named predicates, the guards, collectively denoted by G(v), and (3) a generalized substitution denoted by S(v). An event can perfectly retrace a rewriting-rule of a distributed algorithm [8]. Roughly speaking, it can express through its guard the triggering condition of a rewriting-rule and through its substituting section, the rewriting-rule action. In the event component, an initialization event is a special event that allows to define an initial situation for a model. However, B2Visidia don't translate this event since Visidia enable two initialization categories: global and local initialization. For global initialization, B2Visidia allows to submit an initial value to each label by using the "$Instantiate$" command of the tool. Nevertheless, for a global initialization associated with a local initialization, the designer can differentiate manually the particular nodes, by using the Visidia interface.

- Event guards

Firstly, the two adjacent nodes involved in the computation step are declared in the guard. Then, predicates are defined to express conditions of the rewriting rule. Therefore, a condition can be expressed in many syntactically different forms. For example, the condition $lab(x) = A$ can be written as $x \mapsto A \in lab$. An interesting question arises : Which is the most suitable syntax to express a condition ?
Putting forward the simplicity of the translation processes, we have chosen the first form to represent an *<guard_head>*:

<guard_head> ::= <visidia_variable> 'p_a_c' 'ident' 'p_a_c'
| <visidia_variable> 'p_a_c' 'ident' '↦' 'ident' 'p_a_c'

We denote by *'p_a_c'* a series of brackets, braces or brackets and by *<ident>* a character string. Here, it corresponds to the local variable which can be either a node or an edge. The second syntax of *<guard_head>* is reserved for specifying an edge state. Formally, an event guard can be simple or complex (conjunction or disjunction of predicates).

<event_guard> ::= <literal>
| <literal> ∧ <event_guard>
| <literal> ∨ <event_guard>
<literal> ::= <atomic_predicate>
| '¬' <atomic_predicate>
<atomic_predicate> ::= 'p_a_c' <event_guard> 'p_a_c'
| <expression1> <relop> <expression>
<expression1>::= <guard_head>
| 'card' 'p_a_c' 'g' 'p_a_c' 'ident' 'p_a_c'
| 'ID' 'p_a_c' 'ident' 'p_a_c'

The set of explicit, translatable and deterministic relations operators are: *<relop> ::= '>' | '<' | '≤' | '≥' | '=' | '≠'*

As an example, with the B2Visidia language, we can express the following: $Lab[\{x\}] = \{N\} \wedge \neg Lab(y) = A$. This guard checks if the label of the node x is equal to N and the label of its neighbor y is different from A.

- Event substitutions

There are three kinds of generalized substitutions for expressing the transition associated with an event: (1) the deterministic substitution, (2) the empty substitution, and (3) the non-deterministic substitution [4]. Only the deterministic substitution can be translated by the B2Visidia tool. Syntactically, a substitution is given by <event_substitution>.

<event_substitution> ::= <guard_head> ':=' <expression>
| <visidia_variable> ':=' <visidia_variable> '⩤' 'p_a_c' <new_state> 'p_a_c'
<new_state> ::= 'ident' '↦' <expression>
| 'ident' '↦' <expression> ',' <new_state>

The first form of <event_substitution> is simple, it expresses a relabeling of one edge or one node. The second form allows several labels changing. For example, *Mark (s1 ↦ s2)* := TRUE means that the new label of the edge (s1 ↦ s2) is TRUE and *lab* := *lab* ⩤ {*s1 ↦ A, s2 ↦ A*} means that the new label of s1 and s2 nodes is A. We recall that overwriting as defined in [13] is the relation comprising elements of R2 and of R1 the first element of which does not belong to the domain of R2 when R1 ⩤ R2. In other term, the elements of R2 in (x ↦ a) notation overwrite any elements (x ↦ b) in R1.

We assert that $Machine_2$ of the proposed example needs only two annotations (synchronization and type annotations) to be compliant with B2visidia language and correspond to the $Machine_V$ model.

B. Translation to a Java code

The translation follows a linear process. First, we start by extracting the synchronization annotation and the $visidia\_variable$, then we translate events. The synchronization annotations define the structure of the generated Java code. Concretely, we distinguish three different patterns of Java code for Visidia (a pattern for each synchronization type). For LC0 algorithms, events are placed after achieving an exchange of labels between the two synchronized nodes. For LC1 algorithms, only labels attached to the center of the star can be modified. A control structure has to be added in the resulting code: if the node which implements the algorithm is the center of the star, the event is executed after receiving the labels of its neighbors, else the node is supposed only

to send its label to the center node. Finally, LC2 pattern is similar to LC1, except that LC2 events can be executed by the neighbors too. The synchronization algorithm implementation is recovered from the Visidia API which provides many methods to help the user writing the Java code.

In Visidia, the same algorithm is assigned to each process (each node). Each node has a whiteboard and an editable set of its properties. The set of the node properties are encoded in the $algorithm$ class as vertex properties. In order to deal with these properties, Visidia offers many property methods (for instance $object\ getproperty()$ to retrieve the label of the node which executes the algorithm and $void\ putProperty()$ to add a new property to the node). The $visidia\_variable$ allows to define these properties and this depends essentially on the annotation types. The whiteboard saves the received messages. To implement a whiteboard, we declare the $neighborValue$ variable which will contain the value(s) of the neighbor label(s).

For LC0 algorithms, an event is translated to one or two "if" condition structures, and this depends on the number of nodes who change their states. In the $RelabelingRule$ event of Section IV-A2, the label modification is performed only by the node "$s2$". Then, when we translate the event we take the "$s2$" position (we consider "$s1$" as a neighbor). The translation of the $RelabelingRule$ event is presented in Fig.3.

```
/* Translation of grd1 and grd2 */
    if ((neighbourValue.equals("A"))
        && (((String) getProperty("label")).equals("N")))

/* Translation of act2 and act3 */
      { putProperty("label",new String("A"));
        setDoorState(new MarkedState(true)); }}}
```

Fig. 3. $RelabelingRule$ translation

Finally, to create a new algorithm for Visidia tool, we create a new class in a new file. The new algorithm extends the API classes, depending on the communication mode (synchronous or asynchronous) and the process type (fixed processes or mobile sensors). For our case, the generated code extends the $Algorithm$ class and has the same name as the Event-B machine. For translating supported symbols of B2Visidia to their equivalent in Java language, we have defined some translation rules (see Table I).

VI. Conclusion

In this paper, we have presented B2Visidia, a general framework for producing automatically a Java code from an Event-B specification of a distributed algorithm. We have described its theoretical basis and gave an overview of the implementation. We have presented the B2Visida langage which can be used for a large class of distributed algorithms. We have developed some examples and many other examples have been investigated and gathered in a library of proved programs for distributed algorithms.

We have developed the B2Visidia tool with Java. This tool can be executed locally or online as a Java applet on the web site *visidia.labri.fr*. It offers many services : editing formal specification, importing Rodin files, displaying the resulting Java code, compiling the code, loading old algorithms,.. We are currently working on integrating this tool to the Visidia platform.

Langage *Event-B*	Java language	Comments
x = y	x == y ou x.equal(y)	Condition
x $\neq$ y	x != y	Condition
x > y	x > y	Condition
x < y	x < y	Condition
x $\leq$ y	x <= y	Condition
x $\geq$ y	x >= y	Condition
$\neg$ (Condition)	!(Condition)	Condition
(Condition) $\vee$ (Condition)	(Condition) || (Condition)	Condition
$\forall$ variables · declaration $\Rightarrow$ Condition	for (visit neighbors) {Condition}	Condition
(Condition) $\wedge$ (Condition)	(Condition) && (Condition)	Condition
Lab(x) := y	this.PutProperty(y) ;	Affectation
Lab := Lab $\Leftarrow$ { x $\mapsto$y }	this.PutProperty(y) ;	Affectation
Lab(x) := y + z	this.PutProperty(y + z) ;	Affectation
Lab(x) := y - z	this.PutProperty(y - z) ;	Affectation
Lab(x) := y * z	this.PutProperty(y * z) ;	Affectation
Lab(x) := y $\div$ z	this.PutProperty(y / z) ;	Affectation
ID(x)	*Integer getId()*	Visidia method
card(g(x))	*int getArity()*	Visidia method

TABLE I
Event-B to Java translation syntax

References

[1] H. Attiya and J. Welch, *Distributed computing*. McGraw-Hill, 1998.
[2] G. Tel, *Introduction to distributed algorithms*. Cambridge University Press, 2000.
[3] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.
[4] J.-R. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
[5] L. Voisin, "Public versions of basic tools and platform," ETH Zurich, Public Document Project IST-511599, 29 October 2007.
[6] I. Litovsky, Y. Métivier, and E. Sopena, "Different local controls for graph relabelling systems," *Mathematical System Theory*, vol. 28, pp. 41–65, 1995.
[7] M. Bauderon and M. Mosbah, "A unified framework for designing, implementing and visualizing distributed algorithms," *Graph Transformation and Visual Modeling Techniques (First International Conference on Graph Transformation)*, vol. 72, no. 3, pp. 13–24, 2003.
[8] M. Tounsi, M. Mosbah, and D. Méry, "Proving distributed algorithms by combining refinement and local computations," *Automated Verification of Critical Systems*, vol. 35, 2010.
[9] S. Wright, "Automatic generation of c from event-b," in *Workshop on Integration of Model-based Formal Methods and Tools*, February 2009.
[10] D. Méry and N. K. Singh, "Automatic code generation from event-b models," in *Proceedings of the Second Symposium on Information and Communication Technology*, ser. SoICT '11. New York, NY, USA: ACM, 2011, pp. 179–188.
[11] A. Edmunds and M. Butler, "Tool support for event-b code generation," in *WS-TBFM2010*, February 2010.
[12] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles, "Tom manual," INRIA CNRS, Technical Report, 2008.
[13] J.-R. Abrial, *The B-book: assigning programs to meanings*. New York, NY, USA: Cambridge University Press, 1996.
[14] R. J. R. Back, "On correct refinement of programs," *Journal of Computer and Systems Sciences*, vol. 23, no. 1, pp. 49–68, August 1981.
[15] M. Tounsi, A. Hadj Kacem, M. Mosbah, and D. Méry, "A refinement approach for proving distributed algorithms : Examples of spanning tree problems," in *Integration of Model-based Formal Methods and Tools - IM_FMT'2009 - in IFM'2009*, Düsseldorf Allemagne, 02 2009.