Deliverable 5
Tools

June 29, 2010:10:13 A.M.

Projet RIMEL

ANR-06-SETI-015

Equipe MOSEL, LORIA, Université Henri Poincaré Nancy 1
ClearSy
LABRI, Université de Bordeaux & CNRS
http://rimel.loria.fr
Années 2007-2008-2009

Avertissement The following report has been written by Thierry Lecomte and Mohamed Mosbah.

Contents

List of Figures

Chapter 1

Introduction

This document focuses on the tools developped within the framework of the project Rimel, supporting modelling activities described in Deliverables 1 to 4.
The tools described in this document are:

- **Bart**: a tool that refines (semi-)automatically B models. Automatic refinement has been initally experimented by Siemens Transportation Systems, for the development of safety critical software. To date the biggest implementation is the automatic pilot of the Val de Roissy shuttle: around 80% of the models used for the code generation have not been handwritten, leading to 200 000 lines of Ada code. Bart appears to be a good dissemination vector for refinement in the industry as well as in the academic world.
- **Visidia**: a tool that produces a Java code from an Event-B specification of a distributed algorithm. Distributed algorithms provide an essential framework for understanding computing systems in a range of areas. With Visidia, algorithm design gains precision and validation by proof as well as simulation capabilities.

Chapter 2

B Automatic Refinement Tool

2.1 Introduction to Bart

BART is a tool that can be used to automatically refine B components. This process is rule based so that the user can drive refinement. Its own rule language has been defined in this purpose. It has been designed to be a stand-alone tool, but it may be launched from Atelier B user interface. As an input, Bart must be given at least the machine or refinement to refine. There must be exactly one component to refine.
Automatic refinement is a rule based refinement process for B components (abstractions or refinements). The tool is given a component, and it searches, for each element to refine, some rules that specify how it must be treated. These rules allow to implement design patterns for B models. Bart is specialized for B software models but is likely to be extended to support event B models.

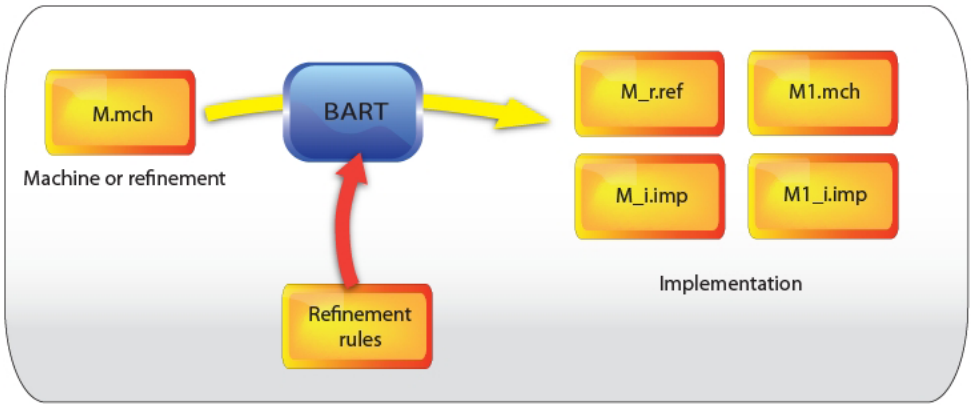


Figure 2.1: Bart: principles

Bart produces a set of B components corresponding to the implementation of this input machine or refinement, by using refinement rules and possibly annotations added to the input machine, to rewrite it. Bart can be considered as a pattern-matcher, as refinement rules are defined in term of patterns and define the way matching terms are transformed. Refinement rules are applied repeatidly applied, generating new machines and/or implementations til one of the two following conditions hold:

- generated components correspond to a translatable B0 implementation. The refinement process is considered as a success.
- No rule can be applied any more. In this case, BART generates machines and display an error message. Two different situations may arise:
  - no rule is applicable: the tool stops silently. The user is expected to add/modify existing rules in order to complete the refinement process.

– a variable needs to be implemented in several components: constraints due to decomposition and implentation may leads to have conflicts (both the model and the refinement rules need to be modified). In this case, a xml file is generated (see figure 2.1).

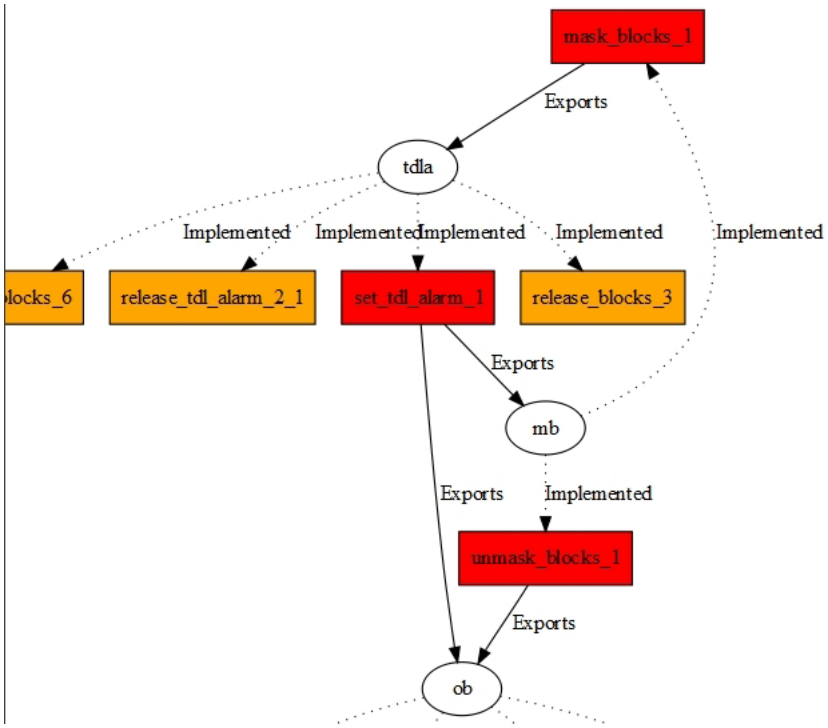


Figure 2.2: Conflicts when refining automatically

The automatic refinement proces requires some specific conditions to be efficient, as demonstrated in figure 2.3:

- SETS and CONSTANTS have to be sperate from the model to be refined (configuration machine)
- similarly, setters and getters have to be separate (inputs machine)

2.1.1 Refined Elements

First elements processed by the Bart tool are the abstract variables of the component to refine (content of the $ABSTRACT\_VARIABLES$ clause).
The tool must produce, for each one of them, one or more abstract or concrete variables that implement it.
Bart processes operations of given component in order to refine them. It must produce, for each operation, a substitution body concrete enough to be put in the component implementation. Refined operations are considered for the whole component abstraction. It means that Bart refines the most concrete version of each operation.
Bart also refines content of initialisation clause of given component. Typically, it produces a concrete result by specifying initialisation substitutions for concrete variables refining content of $ABSTRACT\_VARIABLES$ clause.
Abstract variables are refined first, as other parts of the process need its output to find suitable rules for operations and initialisation. It is necessary at these steps to know how variables have been refined. This variable information is stored as predicates in Bart hypothesis stack.
As it will be described later, refinement process uses rules to determine how each element is refined. A same rule can apply for several elements, so it must be general. In this purpose, the rule language uses jokers, so that rules can contain variable parts.

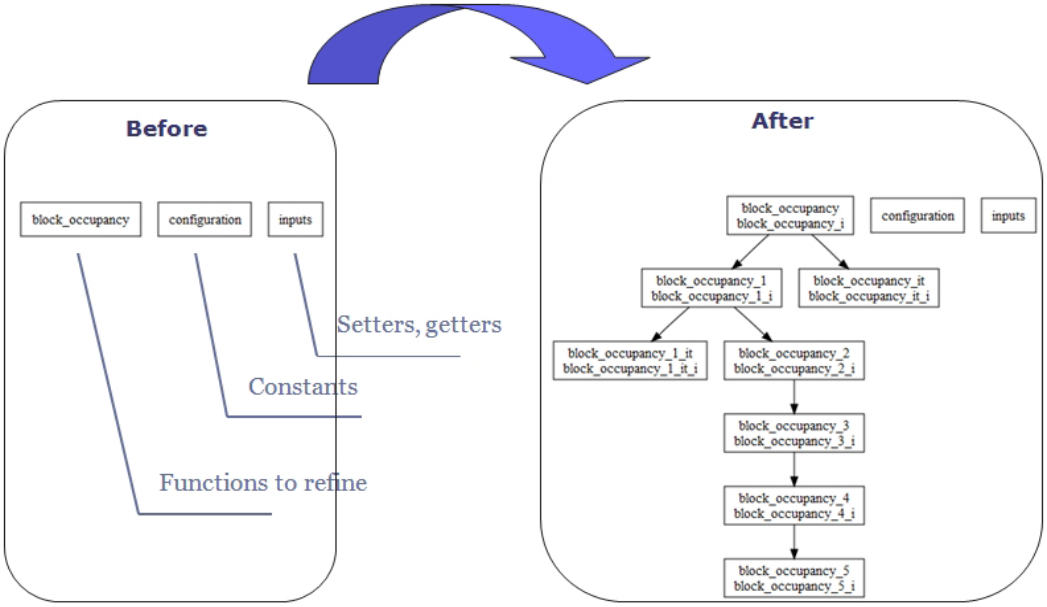


Figure 2.3: Model transformation with Bart

2.1.2 Refinement Rules

Bart uses rules for refining variables, operations and substitutions. These rules belong to different types:

- variables rules, or
- substitution rules, which can be used for both operations and initialisation.

Rules of same type are gathered in theories (ako packages). Theories are associated to predicates/expressions families, as referred in the tactics theory. Rules usually contain a pattern, and may contain a constraint. These two elements are used to know if a rule can be applied to refine a certain element. Rules also contain clauses that express the refinement result.
Rules may have constraints, expressed in their $WHEN$ clause. A constraint is a predicate, which may contain jokers. It may be a complex predicate, built with "&" and "OR" operators. Bart contains a stack of hypothesis , which is built from the machine to refine and its environment. A constraint is successfully checked if its elementary elements (element not containing "&" or "OR") can be pattern-matched with a predicate of the stack so that the complex constraint is true.
According to operators, Bart uses backtracking to try every combination of instantiation that should be a success. If several instantiations can make the constraint be successfully checked, Bart uses one of them. In this case, it is better to write a more detailed constraint to have only one result. If there are several results, Bart could choose one which is not the one the user had planned. Usually, when checking a constraint, some jokers have already been instantiated.
Guards are special predicates which may be present in rule constraint clauses. They allow checking some properties on elements to refine and their environment.
There are two kinds of guards:

- some are simply present in the predicate stack. They are added at the environment loading. For instance $ABCON$ (abstract constant) and $ABVAR$ (abstract variable) belong to this kind of guards.
- the other kind is calculated guards. For these ones, during constraint checking, Bart doesn't try to match them) with the stack, but directly calculates if the guard is true or false. These kinds of guards may also have side effects. For example bnum (numeric test) or bident (identifier test) are calculating guards.

Guards are simply put in the constraint as regular predicates.

This process is used for variables, operations and initialisation refinement, although it is simpler for variables. Every rule contains a pattern. First Bart tries to match it with the element to refine. If it succeeds, it tries, if a rule has a constraint clause, to match the predicate against hypothesis. When checking the constraint, some jokers have already been instantiated by pattern matching. If the constraint checking is a success or the rule had no constraint, then it will be used to refine current element.
Variable process is simpler as variable rules have simple pattern, which is a single joker. Variable rule patterns are only matched in order to instantiate the joker representing currently refined abstract variable. This joker is reused in WHEN or result clauses.

2.2 Refinement Rules Database

Rule files are files containing theories, each theory containing one or several rules used to refine given component. Rule file extension is usually .rmf. A rule file can contain variable, operation, structure and initialisation theories. It can also contain utility theories such as tactic, user pass, or definition of predicates synonyms.
The rule file syntax must also respect certain constraints:

- User pass can be present at most once
- Tactic can be present at most once
- Predicate theory can be present at most once

Order between theories has no syntactical impact, expect for predicates theory: it must be defined before its elements are used in the rule. Order between theories has an impact on the rule research, as the standard process (no user pass or tactic) reads theories from bottom to top. User pass and tactic can be defined anywhere in the file, even before theories they refer to have been defined.

```
RULE assign_a_bool_subset_b_c_11
REFINES
    @a := bool(@b <: @c-@d)
REFINEMENT
    @a := bool(@b <: @c & @b /\ @d = {})
END;
```

Figure 2.4: Example: rule n°1

The Bart tool comes with a set of predefined rule base, contained in the file PatchRaffiner.rmf present in Bart distribution. It provides rules that permit to refine most of the classical B substitutions.

```
RULE assign_a_bool_belongs_b_c_16
REFINES
    @a := bool(@b|->@d : @c*@e)
REFINEMENT
    @a := bool(@b:@c & @d:@e)
END;
```

Figure 2.5: Example: rule n°2

The classical automatic refinement scheme is the following: most elements of given component can be refined using the rule base. If an element can not be refined with it, or needs a more specific treatment, user should write suitable rules in rmf files that will be provided after the rule base on command line, or in the component associated rule file.

2.2.1 Variables refinement

Variable rule research is different from rule research for operations and initialisation. Instead of processing each variable and finding a suitable rule for it, it processes each rule of considered theories (all variable theories or a subset if tactic or user pass is used) and checks if it can be used to refine some variables. This

is necessary because a single variable rule can be used to refine several variables. Once a rule has been selected for one (or several) variable, resulting refinement variables can be calculated from its clauses.
The principle of rule research is the following:

- At the beginning the tool considers the set of abstract variables to refine
- It processes every theory that could be used (according to tactic, user pass or neither) from bottom to top. For each theory, the tool processes all rules of theory from bottom to top. For each variable rule:
  - Bart determines which variables can be refined by current rule
  - Refined variables are removed from the set of remaining variables

This process stops when there are not variables to refine anymore, or when all variable rules to consider have been treated. Variable refinement is successful if all variables have been associated with a rule. It is a failure if all rules have been treated and some variables could not be refined.
For a certain rule, Bart determines which variables it can refine as follow:

- The tool tries every combination of values to instantiate joker list of $VARIABLES$ clause. For each instantiation:
  - Bart checks constraint expressed in WHEN clause against hypothesis stack, with jokers of $VARIABLES$ clause instantiated
    - If $WHEN$ constraint could be checked, variables used to instantiate $VARIABLES$ clause can be refined by this rule
    - Variable refined by the rule are removed from set of remaining variables, to be sure they won't be used in following tried instantiation

If current rule has several jokers in $VARIABLES$ clause, there are more combinations to try than for simple variable rules.

2.2.2 Substitutions refinement

Substitution refinement gathers operation, initialisation and structural rules. Operation and structure rules are identical. Initialisation rules are simpler versions of operation rules. Substitution refinement is more complex than variable refinement, as it can be recursive, i.e. result of refinement for a given substitution may have to be refined too. Furthermore, for a given substitution, refinement may need several sub-processes (use of $SUB\_REFINEMENT$ clause or default refinement behaviours for parallel or semicolon). So refinement sub-branches are created and the underlying structure that can be used to represent substitution refinement is in fact a tree (see figure 2.6).

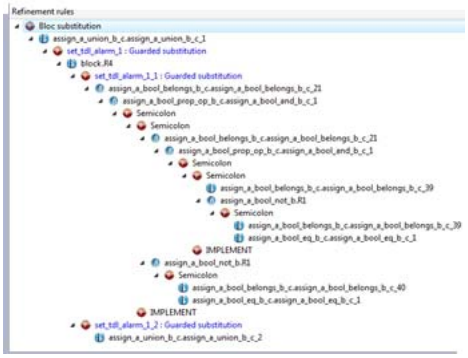
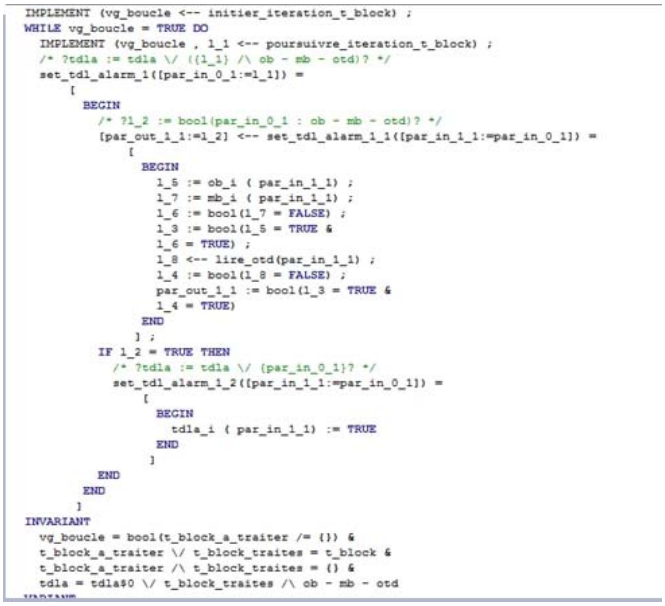


Figure 2.6: Example: refinement tree

The substitution rule research process is simpler than for variables.

- For a substitution to refine, Bart processes each rule file as long as he could not find a rule.

- For each rule file it processes operation theories to consider (all theories, or a subset if tactic or user pass is used) from bottom to top.
- For each theory it processes operation rules from bottom to top
- For each rule, Bart checks if it can be used to refine currently treated substitution.



```
IMPLEMENT (vg_boucle <-- initier_iteration_t_block) ;
WHILE vg_boucle = TRUE DO
  IMPLEMENT (vg_boucle , l_1 <-- poursuivre_iteration_t_block) ;
  /* ?tdla := tdla \/ ({l_1} /\ ob - mb - otd)? */
  set_tdl_alarm_1({par_in_0_1:=l_1}) =
      [
        BEGIN
          /* ?l_2 := bool(par_in_0_1 : ob - mb - otd)? */
          [par_out_l_1:=l_2] <-- set_tdl_alarm_1_1({par_in_1_1:=par_in_0_1}) =
              [
                BEGIN
                  l_5 := ob_i ( par_in_1_1) ;
                  l_7 := mb_i ( par_in_1_1) ;
                  l_6 := bool(l_7 = FALSE) ;
                  l_3 := bool(l_5 = TRUE &
                  l_6 = TRUE) ;
                  l_8 <-- lire_otd(par_in_1_1) ;
                  l_4 := bool(l_8 = FALSE) ;
                  par_out_1_1 := bool(l_3 = TRUE &
                  l_4 = TRUE)
                END
              ] ;
          IF l_2 = TRUE THEN
            /* ?tdla := tdla \/ {par_in_0_1}? */
            set_tdl_alarm_1_2({par_in_1_1:=par_in_0_1}) =
                [
                  BEGIN
                    tdla_i ( par_in_1_1) := TRUE
                  END
                ]
          END
        END
      ]
INVARIANT
  vg_boucle = bool(t_block_a_traiter /= {}) &
  t_block_a_traiter \/ t_block_traites = t_block &
  t_block_a_traiter /\ t_block_traites = {} &
  tdla = tdla$0 \/ t_block_traites /\ ob - mb - otd
```

Figure 2.7: Example: condensed implementation

If each rule file was processed by Bart and no rule could be found for a certain substitution, an operation refinement may occur. Each operation rule has a pattern ($REFINES$ clause) and may have a constraint ($WHEN$ clause).
The substitution refinement process depends, for given rule and substitution, on the presence and content of $SUB\_REFINEMENT$, $IMPLEMENTATION$ and $REFINEMENT$ clauses.
$SUB\_REFINEMENT$ clause contains a comma-separated list of sub-elements. Each sub-element left part is a substitution that may contain jokers. These jokers must all have been instantiated by pattern matching and constraint checking. Right part of the sub-element must be a single and still uninstantiated joker. This clause is used to refine the given substitution and store the result in given joker. This is done before calculation of the rule substitution result, so the sub-refinement can be used to express the result.
$IMPLEMENTATION$ clause expresses the result of current rule. It contains a substitution which may contain jokers. All these jokers must have been instantiated during pattern matching, constraint checking or sub-refinement processing. $IMPLEMENTATION$ clause may also contain concrete operation refinement variable declaration. Using $IMPLEMENTATION$ clause means that given result is the final result of current branch and does not need to be refined again.
$REFINEMENT$ clause expresses the result of current rule. It contains a substitution which may contain jokers. All these jokers must have been instantiated during pattern matching, constraint checking or sub-refinement processing. $REFINEMENT$ clause may also contain abstract or concrete operation refinement variable declaration. Using $REFINEMENT$ clause means that given result is not the final result of current branch. The result of rule must be refined. $IMPLEMENTATION$ and $REFINEMENT$ clauses can not be both used in a same rule. When a rule has been selected (and eventual sub-refinements have been processed), the rule result is calculated by instantiating jokers of its result clause.
A rule can contain both $SUB\_REFINEMENT$ and $REFINEMENT$ clauses. In this case, each subrefinement is calculated and stored in its joker. Then content of $REFINEMENT$ clause is instantiated and refined. For a substitution to refine, if no rule could be found, Bart will check if it can be refined using a predefined behaviour. For some kinds of substitutions, Bart may know how to refine them if no rule is

present. Predefined behaviour can be the end of current branch (skip substitution refinement) or a simple node of refinement tree. In this case, Bart may create one ($BEGIN$ substitution refinement) or several (semicolon refinement) subnodes in refinement tree for current substitution. For each new subnode created by predefined refinement behaviour, the recursive refinement process is restarted as a rule or predefined behaviour will be searched for each one.
The final result (condensed implementation, see an example on figure 2.7) is broken down into several components (machines, refinements and implementations)

2.3 Application

This application chosen to demonstrate Bart comes from one of the B courses, that is used to train students to the writing of specification and its implementation.
A switch is a railway equipment in charge of giving a direction to a train: normal (the train is going straight) or reverse (the train is turning on the left or the right). Knowing the position of a switch is of vital importance as a crash is likely to occur if a train is passing by a moving switch. To do so, three sensors are used to measure the position of the switch (the sensors may return a faulty measure, so three of them are required to achieve the target reliability).
Our objective is to formalize the specification of the function in charge of estimating the position of the switch based on three measurements. Each sensor is returning a measure: normal, reverse or void (void indicating either a moving switch or a faulty sensor). The specification retained is the following:

- If at least one sensor returns normal and one returns reverse, then the estimated position of the switch is void
- Else, if at least one sensor returns normal (resp reverse), then the estimated position is normal (resp reverse)
- Else if, the estimated position is void.

We obtain the following specification:

```
pos <-- measure(m1,m2,m3) =
PRE
  m1 : POSITION &
  m2 : POSITION &
  m3 : POSITION
THEN
  SELECT
    normal : {m1,m2,m3} &
    reverse /: {m1,m2,m3}
  THEN
    pos := normal
  WHEN reverse : {m1,m2,m3} & normal /: {m1,m2,m3} THEN
    pos := reverse
  ELSE
    pos := void
  END
END
```

The handwritten implementation of this specification is obtained by describing the complete algorithm of the estimation. One example is given below:

```
pos <-- measure(m1,m2,m3) =
  CASE m1 OF
    EITHER normal THEN
      IF m2 = reverse or m3 = reverse THEN
        pos := void
```

```
      ELSE
        pos := normal
      END
    OR reverse THEN
      IF m2 = normal or m3 = normal THEN
        pos := void
      ELSE
        pos :=  reverse
      END
    ELSE
      CASE m2 OF
        EITHER normal THEN
          IF m3 = reverse THEN
            pos := void
          ELSE
            pos := normal
          END
        OR reverse THEN
          IF m3 = normal THEN
            pos := void
          ELSE
            pos := reverse
          END
        ELSE
          pos := m3
        END
      END
    END
  END
```

Bart is able to generate automatically the implementation of this specification, but its solution is quite different from the handwritten one. Indeed Bart generates three components: switch_i (the implementation), switch1 (imported by switch_i) and switch1_i (the implementation of switch1).
The switch_i implementation is listed below, in several parts:

```
pos <-- measure(m1 , m2 , m3) =
  VAR
    l_1 , l_2 , l_3 , l_4 , l_5 , l_6
  IN
                    /* Rule : select.select_with_else_split_cond_when */
                    /* Rule : default.if_then_else_0 */
                    /* ?l_1 := bool(normal /: {m1 , m2 , m3})? */
                    /* Rule : default.default */
    l_1 <-- measure_1(m1 , m2 , m3) ;
```

Rule applications and refined substitutions appear as comments in the generated model above. These comments are removed from the remaining model below to ease its reading. The structure of the algorithm is explicit, but the calculation of the position is performed through imported operations measure_1 to measure_13 that are specified in the machine switch1.

```
    IF l_1 = TRUE THEN
      l_2 <-- measure_2(m1 , m2 , m3) ;
      IF l_2 = TRUE THEN
        pos <-- measure_3
      ELSE
        l_3 <-- measure_4(m1 , m2 , m3) ;
        IF l_3 = TRUE THEN
```

```
          l_4 <-- measure_5(m1 , m2 , m3) ;
            IF l_4 = TRUE THEN
              pos <-- measure_6
            ELSE
              pos <-- measure_7
            END
        ELSE
          pos <-- measure_8
        END
      END
    ELSE
      l_5 <-- measure_9(m1 , m2 , m3) ;
      IF l_5 = TRUE THEN
        l_6 <-- measure_10(m1 , m2 , m3) ;
        IF l_6 = TRUE THEN
          pos <-- measure_11
        ELSE
          pos <-- measure_12
        END
      ELSE
        pos <-- measure_13
      END
     END
   END
END
```

switch1 is a staless machine, meaning that no variables are modelled: operations are returning a value obtained by combinatorial combination of input parameters.
Here we only describe specification and implementation of the operation measure_9. Its specification is given below:

```
par_out_0_1 <-- measure_9(par_in_0_2 , par_in_0_3 , par_in_0_4) =
  PRE
    par_in_0_4 : POSITION &
    par_in_0_3 : POSITION &
    par_in_0_2 : POSITION &
    0 = 0 &
    not(not(normal : {par_in_0_2} \/ {par_in_0_3} \/ {par_in_0_4}))
  THEN
    par_out_0_1 := bool(reverse /: {par_in_0_2 , par_in_0_3 , par_in_0_4})
  END
```

Bart has generated the preconditions of the operation:

- typing predicates of input parameters par_in_0_2, par_in_0_3 and par_in_0_4
- the ELSE condition of the test bool(normal /: m1 , m2 , m3) == TRUE)

The substitution

```
  par_out_0_1 := bool(reverse /: {par_in_0_2 , par_in_0_3 , par_in_0_4})
```

is implemented as

```
VAR
  l_2 , l_3 , l_4 , l_5 , l_6
IN
  l_5 := bool(reverse = par_in_0_2) ;
  l_6 := bool(reverse = par_in_0_3) ;
```

```
  l_3 := bool(l_5 = TRUE or l_6 = TRUE) ;
  l_4 := bool(reverse = par_in_0_4) ;
  l_2 := bool(l_3 = TRUE or l_4 = TRUE) ;
  par_out_0_1 := bool(l_2 = FALSE)
END
```

The final model is a bit more verbose than the handwriten one. The automatic refinement process induces more steps and decompositions, in order to keeep the proof at a reasonnable difficulty. On this small example, the size of the binary executable of the operation measure is 555 bytes for the handwritten version, and 2953 bytes for the automatically generated one, by using gcc compiler without any optimization. This size could be reduced by a smarter C code generator able to inline operations.

2.4 Dissemination

2.4.1 Events

BART has been presented at several occasions, during international conferences and workshops:

- ”Automatic Refinement and Code Generation: leassons learned”, T. Lecomte, Workshop ”Recent Innovations and Applications in B” - FM’2009 - Eindhoven
- ”Applying a Formal Method in Industry: a 15-year trajectory”, T. Lecomte, Formal Methods for Industrial Critical Systems - FM’2009 - Eindhoven
- ”Bart: Automatic Refinement in B”, A. Requet, Workshop ”B Dissemination Day”, Grace Symposium on Advanced Software Engineering - Tokyo
- ”Model Inside”, T. Lecomte, Journées Neptune 2010, Toulouse
- ”Ten years disseminating B”, Teaching Formal Methods: The B Method, Journées scientifiques de Nantes 2010 - Nantes

A one-day tutorial has been given during the ABZ’2010 conference. Around 10 researchers have attended this event. A specific version of Bart was released for that occasion, that will be part of the next Atelier B 4.0.2:

- the refinement engine has been corrected and improved. In particular, simple models are now correctly handled by the tool.
- the refinement rules database has been corrected and completed.

Following this tutorial, we organized a special 3-day session in house for one of the researchers (UFRN, Brazil) willing to evaluate precisely how Bart could be used for developping a smartcard embedded application.
Finally Bart will be presented through an extended session on software development during the Workshop on B Dissemnation, jointly organized in November 2010 with SBMF 2010 (Natal, Brazil).

2.4.2 Courses

Bart is being integrated to B courses in France, namely:

- ”Applications industrielles de B” - IRIT Toulouse - Master 2: course given to software engineers, focusing on industrial applications of B, including automatic refinement
- ”Spécification et conception sécurisées” - ENSI Bourges - 3ème année option sécurité logicielle: complete course on specification and refinement for security software development
- ”Méthodes formelles” - ENSME Gardanne - 3ème année: complete course on specification and refinement, for microelectronics engineers
- ”Développement de logiciels critiques” - ESIL Marseille - 3ème année: complete course on specification and refinement, for software engineers

2.4.3 Ressources

All ressources are available online:

- a dedicated wiki is being run at http://www.tools.clearsy.com/index.php5?title=BART_Project, gathering all public documents
  - Bart Specification (http://www.tools.clearsy.com/index.php5?title=Bart_Specification_1.1)
  - Bart User Manual): http://www.tools.clearsy.com/images/1/16/BART-User_Manual-29.04.2009.pdf
- A source code repository hosted on sourceforge: https://sourceforge.net/projects/bartrefiner

Ressources from the one-day tutorial given in Orford, Canada in 2010 will be made available when Atelier B 4.0.2 is released.

Chapter 3

Visidia

3.1 Introduction to Visidia

ViSiDiA [4, 2, 3] is a tool for the execution and visualization of distributed algorithms. It allows the user to model a network, to implement and to execute a relabeling system (the notion of local computations and in particularly the graph relabeling systems is a framework to encode a distributed system [5]). It is written in Java where the distributed processors are simulated by the Java threads and the interconnection of processes is abstracted by a communication graph model. The provided algorithms implemented in JAVA can be run on top of the selected network.
The architecture of the tool is composed of three main parts, namely the graphical user interface, the simulator and the algorithm library. The graphical user interface of the tool is a graphical environment that allows the user to import or to draw a network easily. The simulator allows visualizing the execution of a distributed algorithm. It models a network of asynchronous processors. Each processor communicates only with its immediate neighbors by message passing. Recently, Visidia supports the simulation and the visualization of distributed algorithms in the mobile agent model. An algorithm is implemented by a Java program which will be instantiated on each node of the graph, and executed asynchronously by the corresponding processor. A node is implemented by a class that contains its identifier, its internal state, its degree, and optionally the size of the graph. The tool provides a library of high level primitives to program the corresponding local computations. In order to implement distributed algorithms specified by local computations, three kinds of local computations are considered :

- RV (Rendez-Vous): in a computation step, the labels attached to a couple of nodes connected by an edge are modified according to some rules depending on the labels appearing on the edge, and on its nodes.
- LC1 (Local Computation of type 1): in a computation step, the label attached to the centre of the star (We call a star, a node together with its neighbors. We refer to these neighbors as the leaves of the star) is modified according to some rules depending on the labels of the star (labels of the leaves are not modified).
- LC2 (Local Computation of type 2): in a computation step, labels attached to the centre and to the leaves of the star may be modified according to some rules depending on the labels of the star.

After getting a transient synchronization with one neighbor or with all its neighbors, a processor tries to perform a rewriting step. It exchanges its labels and attributes with its neighbor(s). If applicable rule is found, the process performs some local computations and updates its labels and its attributes according to the right-hand side of the rule. A rule can be applied if it is consistent with the states of the process and its neighbors.

3.2 The approach principle

The proposed approach is a cyclical approach, it begins with a formal specification of a distributed algorithm and finishes by a correct specification with its implementation which allowing the visualization under

Visidia. In Fig.3.1, we show the global synoptic of the approach as well as its different stages. In our approach we consider the designer as the principal actor, which must occur at all stages. At the beginning he is invited to provide a specification of the distributed algorithm. In doing so, he can use the Rodin platform to be sure of the syntax and proofs correctness of his specification. Nevertheless, the input file is no fit to be directly translated; for this reason B2Visidia is equipped with a filtering and rewriting mechanism which allows to retrieve and transform the Rodin file to be adapted for the translation.
However, this step can be considered as optional, because the designer can directly introduce its specification into B2visidia tool. Once the source file has been filtered and rewritten (or the specification is writing directly under B2visidia) the designer can translate the specification to obtain a suitable Java code for Visidia. Finally, he can launch Visidia to test the specification;
here two possibilities are considered:

- If the test is successful, then he can conclude that the specification is a correct representation of the algorithm.
- Otherwise, the designer can locates the potential problem and so he have to re-start the process from the beginning and corrects the specification where the problem lies. This operation can be done when the simulation is completed and judged conform to the waitings of the designer.

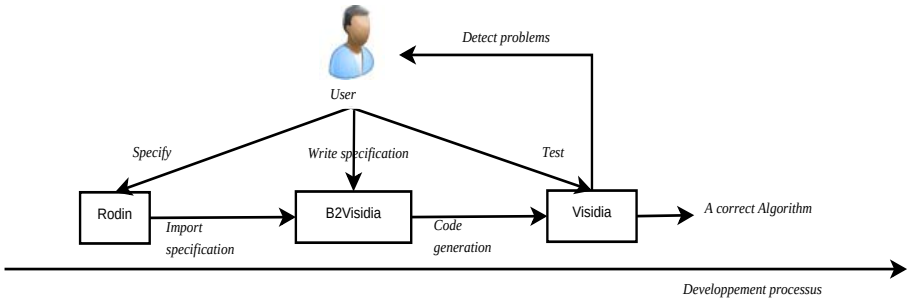
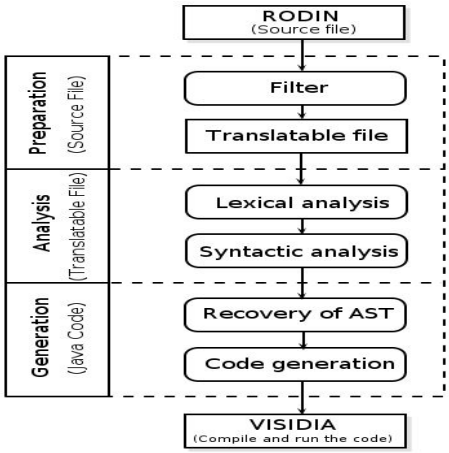


Figure 3.1: The graphical user interface of B2Visidia

In this report, our work will be focused on the translation step by presenting in more details the outline of B2Visidia.

3.3 Global architecture

Translating an Event-B model in a concrete language (such as Java language) is not possible in one shot. Since, Event-B specifies any system without taking into account its implementation. B2Visidia makes it possible to translate an Event-B specification (with a few added annotations) into a Java code for Visidia. As presented in the following figure, our approach includes three stages: the initial step consists in preparing the source file stored as an XML file in the Rodin platform. The goal is to generate a simple and translatable text file that holds the useful parts for the translation.



To this end we have chosen Tom [1] which is a language and a software environment suitable for programming various transformations on trees/terms and also it can be used to match and rewrite XML documents.

Once the file has been re-written, we perform, in the second step a lexical and a syntactic analysis of the translatable file to build an Abstract Syntax Tree (AST) of the algorithm specification. Finally, we generate a code for Visidia. In this step, we use suitable rules to perform the conversion of AST nodes and generate the corresponding Java code.

3.4 Case study

The purpose of this case study is to illustrate how B2Visidia will translate a distributed algorithm specification to provide a java code for Visidia tool. In this case study we will show an example of the spanning tree algorithm implemented with the LC0 synchronization.

3.4.1 Algorithm description

We consider an algorithm called Spanning Tree: Distributed Computation without explicit termination. It may be encoded by the graph relabeling system R=(L,I,P) that is defined by the set of the possible labels that describe processor status L={N, A, 0, 1}, the initial state I={N, A, 0}, and the set of relabeling rule P={R1}. Initially, we suppose that a unique "active" node has an A-label state, all other "neutral" nodes having "N" label and all edges are in a "passive" state (represented by the "0" label). At any step of the computation, an active node may activate one of its neutral neighbors and mark the corresponding edges which gets the label "1". This computation stops as soon as all the processors have been activated. The spanning tree is then obtained by considering all the links with "1" label. An elementary step in this computation may be depicted as a relabeling step by means of the relabeling rule R1, given in Figure 3.2, which describes the corresponding label modifications:

Figure 3.2: The relabeling rule R1

Formally, the chosen algorithm can be specified with the refinement technique in several models and by using the Event-B method. The last concrete model that will be translated by our tool is detail in the next section.

3.4.2 Algorithm specification

We assert that the last model in the specification contains three events: initialization, span and rule. The span event computes the solution in one shot; it indicates that the tree can be computed if there does not exists a node with a "N" label. When it occurs, the graph is considered irreducible (no rule can be applied), a spanning tree is computed and the execution of the algorithm is finished. Formally this event is represented as follows :

EVENT span

REFINES span

WHEN

$grd2 : lab^{-1}[\{N\}] = \varnothing$

THEN

$act1 : st := new\_tree$

END

The computed variable "st" contains the resulting spanning tree. "lab" function assigns to each node a label and "Mark" function attributes a label to each edge. Formally, these variables are defined by the following invariants.

INVARIANTS

inv1:

lab $\in$ ND $\rightarrow$ label

inv2:

st $\in$ ND $\leftrightarrow$ ND

inv3:

Mark $\in$ g $\rightarrow$ {0,1}

"Rule" event simulates the elementary computation step of each node in the graph. It contains exactly 4 guards and 3 actions. The first and the second guard mean that the node s1 is activated and its neighbors are not yet. Through the third guard, it is understandable that s1 and s2 are neighbors. The two last actions specify that the corresponding state changes according to the algorithm rule.

EVENT Rule

REFINES Rule

ANY

$s1$
$s2$
u

WHERE

$grd1 : lab[\{s1\}] = \{A\}$
$grd2 : lab[\{s2\}] = \{N\}$
$grd3 : s1 \mapsto s2 \in g$
$grd4 : s1 \mapsto s2 = u$

THEN

$act1 : new\_tree := new\_tree \cup \{s2 \mapsto s1\}$
$act2 : lab(s2) := A$
$act3 : Mark(u) := 1$

END

The first step in the translation processes is to extract all visidia_function as well as its types declared in the invariant component. Types are divided in two families: The first describes the node state and the second expresses the state of the edge (for the moment it is not useful for Visidia). For our example, we assert that only "lab" and "Mark" are two visidia_function where the first one is intended to node state (having "String" as type) and the second one to edge state. In accordance with these functions, a rewriting step of the source file is performed and a translatable file is generated. This latter consists in a machine name, a synchronization type; a variable list contains names of visidia_function and the "Rule" event. This event will be transformed too and it will keep the list of local variables, the first two guards and the last two actions because they give rise to a state change. After that, the translatable file will be analyzed and an AST will be generated. Finally, according to some translation rules, as they are presented in the previous section, a Java file will be created. The following Java code structure is a result of the automatically translation generated by the B2Visidia.

```
package visidia.algo;
import visidia.simulation.*;
import visidia.misc.*;
import java.util.*;

/* Java Class Declaration (we use the same name of the specification)*/
public class Spanning_Tree_RDV extends Algorithm {

static MessageType synchronization = new MessageType("synchronization", false, java.awt.Color.blue);
static MessageType labels = new MessageType("labels", true);
```

```
    public Collection getListTypes(){
        Collection typesList = new LinkedList();
        typesList.add(synchronization);
        typesList.add(labels);
        return typesList;}

    public void init(){
        int graphS=getNetSize();
        int synchro;
        boolean run=true;
        String neighbourValue;

        while(run){
            synchro=synchronization();

/* Exchange of states (send and receive of messages) */
            sendTo(synchro,new StringMessage((String) getProperty("label"),labels));
            neighbourValue=((StringMessage) receiveFrom(synchro)).data();

/* translation of grd1 and grd2 of Rule event */
    if ((neighbourValue.compareTo("A")==0) && (((String) getProperty("label")).compareTo("N")==0))

/* translation of act2 and act3 of Rule event */
      {
        putProperty("label",new String("A"));
        setDoorState(new MarkedState(true),synchro);
       }}}

/* we define methods needed for LCO synchronization (imported from Visidia API)*/
    public int synchronization(){...}
    private int trySynchronize(){...}
    public void breakSynchro() {...}

    public Object clone() { return new Spanning_Tree_RDV(); }
}
```

3.5 The B2Visidia tool description

At this stage, B2Visidia tool is operational and offers an easy way to generate a Java implementation of a large class of distributed algorithms specified in Event-B. It is developed in Java and provides several services. The B2Visidia Graphical User Interface (GUI) (presented in Fig. 3.3) has been designed in a simple way to provide visibility and better accessibility to users. The GUI is divided into four topic parts:

1. An input pane on the left side: It is a simple editor window that allows to the designer to edit a specification of the last concrete machine by using a subset of Event-B language that is simple enough to write a specification that can be adequately translated. Also it displays the imported file from the Rodin platform after being filtered and rewritten. The input pane contains a scrolling text area where the user may enter a specification in final form.

2. An output pane on the right side that consists in a non-editable scrolling text area where the translation results are displayed.

3. A console pane on the bottom of the window. It is a display device for the tool and it serves to inform the user by means of messages on the current state of the system.

4. A set of icons implementing the most useful functionalities associated to the current view Between the input and the output pane. We describe the icons functionalities in the order in which they have been displayed in the interface :

   (a) Check command verifies if the input specification satisfies the syntax of the defined Event-B grammar subset.

   (b) Instantiate is an optional command that allows to instantiate all the nodes states of the graph depending of the Visidi_function.

(c) Translate is the most relevant command in the tool; it allows to generates a Java file that stores it in the B2Visidia_results directory of the Visidia project.

(d) Visidia command allows launching Visidia under our tool.

(e) File rewrite command appears when a Rodin file is chosen. It rewrites the file and displays the result in the input pane.

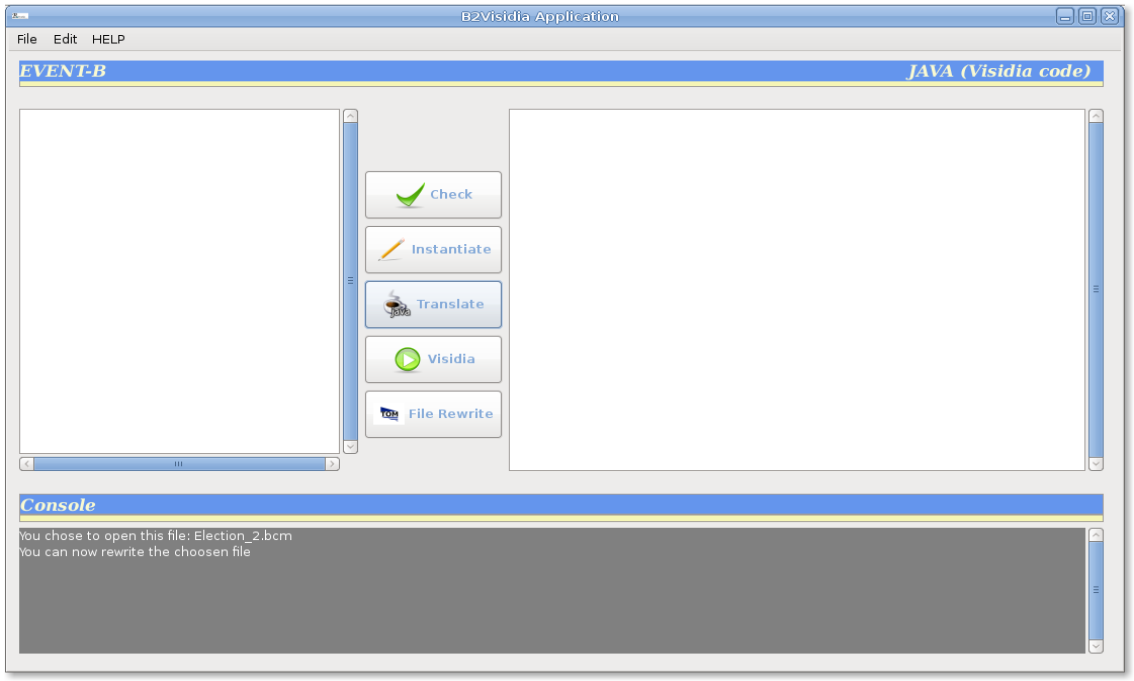


Figure 3.3: The graphical user interface of B2Visidia

Others technical information of our tool is presented in the following table. (To extract these information we have used a ”eclipse metric” plug-in that serves to calculates various metrics of the code during build cycles and warns via the Problems View if it exists)

Metric	Total
Number of packages	16
Number of classes	89
Number of methods	215
Total lines of code	7679

Chapter 4

Conclusion and perspectives

In this report, we have presented the tools developped within the framework of the RIMEL project, namely:

- **Bart**: This tool allows for a B0 implementation for a machine or a sufficiently detailed B refinement to be automatically generated. BART operates on the basis of refinement rules. Additional refinement rules may be added in order to allow for the customization of the refinement of some components. BART has been integrated into Atelier B 4.0. Dissemination in the railways industry is expected when Atelier B gets certified.
- **B2Visidia**: B2Visidia tool that produces a Java code from an Event-B specification of a distributed algorithm. This approach is based on the combination of local computations and the stepwise refinement in the Event-B. We have described its theoretical framework and we have explained how it works by means of an example of a spanning tree algorithm. At this stage, the described components of our approach are operational and the B2Visidia tool is under development with other examples of distributed algorithms.

Bibliography

[1] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom manual. Technical report, INRIA CNRS, 2008.

[2] M. Bauderon, S. Gruner, and M. Mosbah. A new tool for the simulation and visualization of distributed algorithms. *MFI'01*, 1:165–177, mai 2001. Toulouse, France.

[3] M. Bauderon, S. Gruner, Y. Mtivier, M. Mosbah, and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. *Second International Workshop on Graph Transformation and Visual Modeling Techniques, ENTCS*, 50(3):229–239, juillet 2001. Crete, Greece.

[4] Michel Bauderon and Mohamed Mosbah. A unified framework for designing, implementing and visualizing distributed algorithms. *Graph Transformation and Visual Modeling Techniques (First International Conference on Graph Transformation)*, 72(3):13–24, 2003.

[5] I. Litovsky, Y. Métivier, and Éric Sopena. Different local controls for graph relabelling systems. *Mathematical System Theory*, 28:41–65, 1995.