Return of Experience on Automating Refinement in B

Thierry Lecomte[1]

[1] ClearSy,
Aix en Provence, France
Thierry.lecomte@clearsy.com

Abstract. Refining a B specification into an implementation can be a complex and time consuming process. This process can usually be separated in two distinct parts: the specification part, where refinement is used to introduce new properties and specification details, and the implementation, where refinement is used to convert a detailed B specification into a B0 implementation. This article presents experience on the development and use of a refiner tool that automates the production of implementable models, in a number of industrial applications.

Keywords: B formal method, refinement, deployment, industry

1 Introduction

Historically, the B Method [1] was introduced in the late 80's to design correct by construction, safe software. B and Atelier B[1], the tool implementing it, have been successfully applied to the industry of transportation with the development of large size embedded software [2], [3], [4], [10] and to a lesser extent to other application domains [5], [7], [11], [12], [13], [15], [16]. Based on the notion of refinement, the B method allows writing software specification and implementation by using the same formal language. Specification is progressively transformed (refined) into implementation by adding algorithmic details and/or transforming abstract data types into concrete (implementable) ones. If the specification is the result of a purely human activity, the refinement process could be automated as it consists in a sequence of transformations applied to the specification in order to obtain an implementable model.

In this article, we expose our return of experience on applying automatic refinement in industry, through a number of industrial applications. In section 2, an overview of refinement in B is given. Automatic refinement key concepts and tooling are presented in section 3. Section 4 exposes industrial applications of automatic refinement. Finally section 5 concludes and discusses further work.

[1] *http://www.atelierb.eu/*

2 Refining in B

The B method is mainly aimed at software development. B models are structured in components: a component is usually[2] made of a specification model and an implementation model (the implementation being a refinement of the specification).

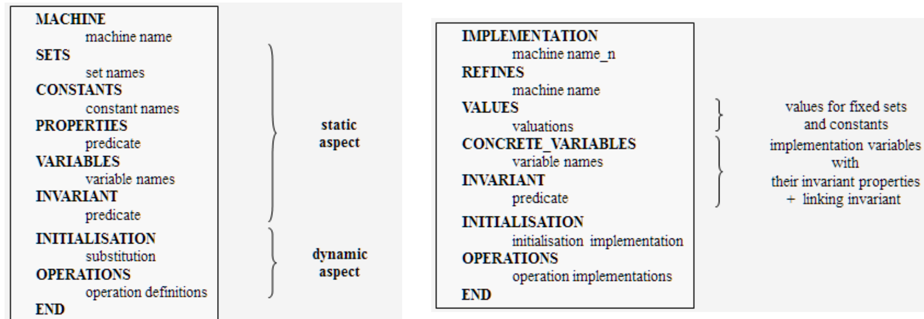


Figure 1: structure of B specification and implementation

An implementation may use other specification models by calling their operations (related models are imported) or by having a read access to the value of variables and constants (related models are seen). A B model is made of variables, constants and operations that modify the variables[3]. Variables and constants have types and constraints linking some variables and constants can be expressed (called invariant for variables and properties for constants). Variables and constants can be either abstract or concrete. Operations are defined with substitutions [8] that can either be deterministic or non-deterministic. Depending of the nature of the model, substitutions used to describe operations are restricted: for example, sequencing is forbidden in specifications, as well as loops, and non-determinism is forbidden in implementations.

Major restrictions on B modelling are:

- [***R1***] The obligation to have a concrete variable implemented in one and only one implementation (it couldn't be modified at two different places). This constraint has a major impact on the structure of B projects: dataflow has to be considered first before defining the components.
- [***R2***] The imports graph must be a tree (see Figure 2): each concrete module except the tree root must be imported in the project to insure that the properties proved locally still hold at global level.

[2] Other cases are: (1) Basic machines have no B implementation as they are used to connect B models with third party software (2) Several successive refinements might be necessary to obtain implementation.

[3] Model initialisation is also considered as an operation. However it can be executed only once and variables properties have to be set after this execution.

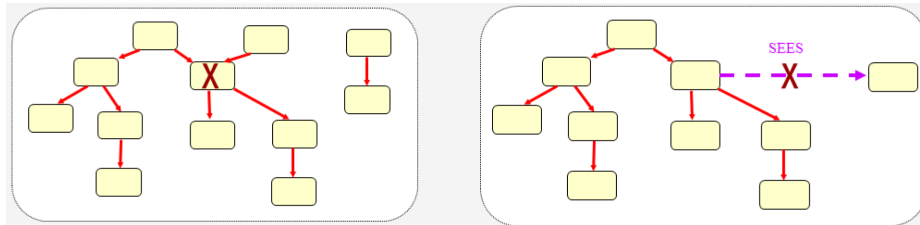


Figure 2: incorrect imports graphs

- [***R3***] The dependency graph must not have any cycle. In this case, it would not have any valid order of initialisation.

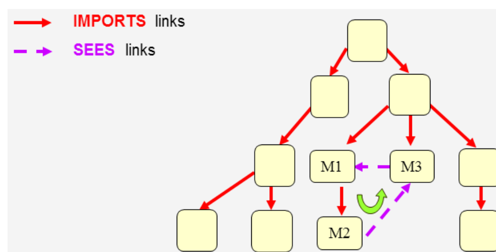


Figure 3: incorrect dependency graph

- [***R4***] An operation defined in the specification of a component can't be used in the implementation of this component. It has to be called in another implementation that imports this component. This constraint requires adding extra-layers in the B project.

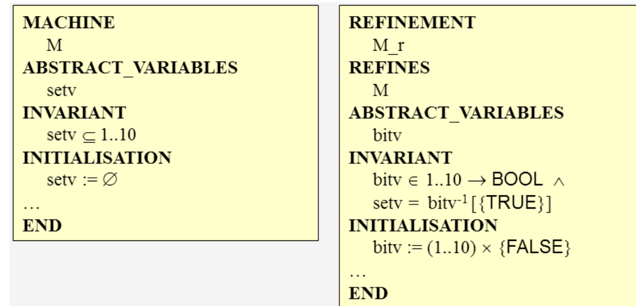
On the other hand, this modular decomposition based on specification importation allows proving the models more easily as it breaks down proof complexity and factorizes the proof of operations. Models are also considered as easier to read and to understand.

As an example, let us consider the example shown in Figure 4. The specification (machine M) declares a variable *setv*, typed as a subset of 1..10. This variable is initialized as empty. Obviously this variable can't be implemented as it is. Sets are often refined by using a table (a function) that associates to each element of the domain (here the interval 1..10) a Boolean value indicating if the element belongs to the set. In this example, the new variable *bitv* is declared as a total function from the interval 1..10 to BOOL.

The gluing invariant

$$setv = bitv^{-1}[\{TRUE\}]$$

establishes a direct relation between the variable *setv* (that doesn't exist anymore in refinement M_r) and the variable *bitv*: all elements for which *bitv* is valued as TRUE belong to *setv*. Initialisation has also to comply with this gluing invariant: *setv* being initialised as empty, *bitv* has to be initialised with all elements associated to FALSE.



```
MACHINE
    M
ABSTRACT_VARIABLES
    setv
INVARIANT
    setv ⊆ 1..10
INITIALISATION
    setv := ∅
…
END
```

```
REFINEMENT
    M_r
REFINES
    M
ABSTRACT_VARIABLES
    bitv
INVARIANT
    bitv ∈ 1..10 → BOOL ∧
    setv = bitv-1[{TRUE}]
INITIALISATION
    bitv := (1..10) × {FALSE}
…
END
```

Figure 4: example of specification and refinement

3 B Automatic Refinement

Rationale. Automatic refinement has been initially imagined [6], developed and put into existence by Matra Transport with the automated "Canarsie line" metro in New-York [9]. By abstract model (see Figure 5), we do not mean the collection of all specification models of the project but a selection of related top level components, including specification, refinement and implementation models, that capture the specification of the software. This specification is abstract but contains all details: it just needs to be transformed into an implementable model.

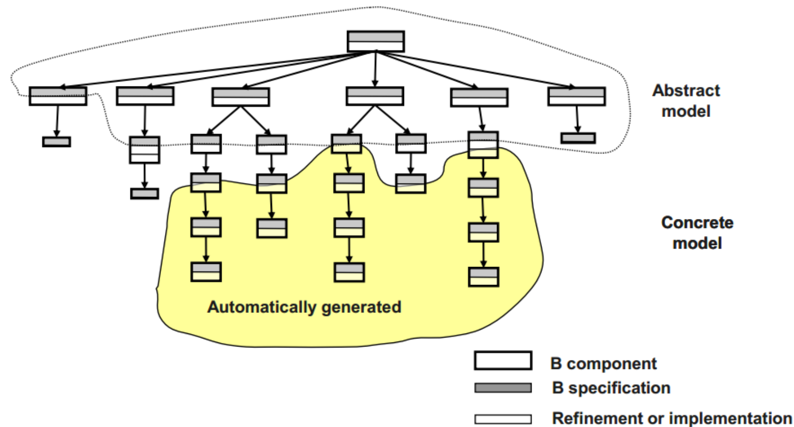


Figure 5: separation between abstract model and concrete model

Automatic refinement is aimed at automatically generating the concrete model (or part of it) from the abstract model, with the objective to dramatically reduce development costs. Indeed, safety critical software usually requires twice the workload because of additional testing, verification and validation. Automatic

refinement is aimed at reducing the workload by up to half as the concrete model is automatically generated and the model is easier to validate. Finally safety critical software would only require the budget of a non-safety critical software development to complete.

Concepts. The main concept is the automatic building of the concrete model by applying refinement patterns step by step, the refinement patterns being expressed as refinement rules. The data (variables) are refined first. The substitutions are then refined.

Abstract model is usually constructed with abstract data typed as set, partial function, relation, etc. As B0 model only allows concrete data type as scalar, total function, etc., data refinement consists in replacing abstract data with more concrete one : this needs to introduce a gluing invariant describing how abstract variable becomes concrete. Data refinement can be done in several steps: not directly from abstract to concrete but with intermediate data representation. Data can already be concrete but be refined in order, for instance, to reduce the memory footprint. In principle, the choice of a refinement for a variable is done regarding its initialisation and operations manipulating this variable. In practice, the choice is done regarding the type of the variable. However types for concrete variables are limited, B0 initialisation is limited too. To automate the data refinement, refinement patterns are given describing the constraints that have to be fulfilled by refined data, the introduced refining variable, and gluing invariant. Then the automatic refiner chooses a refinement pattern regarding the properties of the variables. This choice can be modified by the user who can add new refinement pattern into the tool.

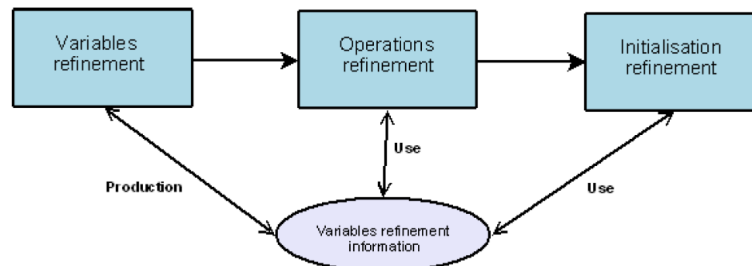


Figure 6: refinement process order

For substitution refinement, treatments on abstract data have to be refined to become translatable treatment on concrete data. B0 allowed substitutions are : assignment, IF, CASE, and WHILE. Similarly to data refinement, substitution refinement is based on refinement patterns that should make explicit under which constraints they can be applied, what are the resulting substitutions, and either it is an implementation or it should be refined one more time.

The refinement of a component consists in the 3 successive actions (see Figure 6): determining refinement pattern for variables of a component, applying refinement patterns for all the operations of a component, and producing the resulting B component. When a variable is refined, related information is stored (variable

refinement information) in order to be reused later on and to speed up the process (variable refinement is elaborated once for all). The recursive application of these 3 steps should lead to a B project fully implemented, if the refinement patterns can be applied at each step of the process. If it is not true, the refinement process stops and some interaction with the user are required to modify existing refinement patterns and/or add new ones.

Tools. In 1997, Matra Transport International (now Siemens Transportation Systems) developed internally a tool called edithB that was used for the development of the Automatic Train Protection System of Canersie Line in New York. This tool was developed in Ada.
In 2008, with Siemens agreement, ClearSy developed a similar tool, called BART (B Automatic Refinement Tool), in order to allow the Community to benefit from automatic refinement tools. The development of the tool has been partly supported by grant No ANR-06-SETI-015-03 awarded by "Agence Nationale de la Recherche" during the R&D project RIMEL (Incremental refinement of event models). In 2009, BART[4] is integrated to the first open-source version of Atelier B (4.0). BART is made more generic than EdithB, in particular the support of the vital coded-monoprocessor is removed. This tool is developed in C++/Qt.

Rule	Operation call	Generated operation
IMPORTED_OPERATION(name => add out => (@a), body => (@a := @e))	aa <-- add1(bb, cc)	out <-- add1(in1,in2) = BEGIN out <-- in1 + in2 END

Figure 7: example of an IMPORTED_OPERATION substitution

For both tools, refinement patterns are expressed using a rich language [9] that allows to specify guards/constraints like DECL_OPERATION (that requires for an operation to be defined), to tune the refinement process and to use advanced substitutions like TYPE_ITERATION (that generates while loops when iterating over a type) or IMPORTED_OPERATION (that generates operation definition and operation call when a treatment needs to be exported to another machine).

```
RULE assign_a_b_6
REFINES
    @a := @b
WHEN
    B0(@a) &
    DECL_OPERATION(@c <-- @d | BEGIN @c := @b END)
IMPLEMENTATION
    @a <-- @d
IMPLEMENT
     @a
END;
```

Figure 8: example of a rule using DECL_OPERATION. The valuation of @a by @b is replaced by the call of an operation (named @d) that returns @b.

[4] https://sourceforge.net/projects/bartrefiner/

These advanced substitutions allow mimicking human refinement by defining precisely how to refine variables and substitutions. These advanced substitutions are replaced by regular substitutions during the automatic refinement process.
The refinement engine manages a stack which contains information about the project, and how variables are refined. This stack also contains tags generated by the refinement rules when applied, these tags have no prior semantics but could be used by refinement rules to give a particular direction to the refinement process.

Metrics. Detailed metrics on edithB are not publically available. Concerning BART, the complete software is made of 264 kloc decomposed as follows:

- Core tool: 89 kloc.
- GUI: 12 kloc
- B Compiler : 154 kloc
- Bwidgets : 7 kloc.

The core tool is made of the refining engine (the biggest module – 7 kloc), the rules parser, the splitter, the namer and the pattern matcher. The complete development was completed in 10 months.

The tool was tested and validated using regular test bench. Indeed, even if the tool is directly contributing to the building of SIL4[5] software, the tool is not expected to be "correct-by-construction" – in case of design error, the generated model should be not provable. In this case, the Atelier B prover is our safety belt.

4 Industrial Applications

Several applications have been developed with edithB and BART.

EdithB has been used for an automatic train protection software (embedded software in charge of stopping a metro in case it is not able to comply with speed limit) in New York [10] and for a wayside control unit (in charge of avoiding train collision) in Roissy airport [2]. If we compare the figures from a manual development [3], [4] and from an automatic refinement development, for the same kind of application, it is apparent that automatic refinement is generating more lines of B models and target code (see Table 1). Meteor and Canarsie being similar applications, automatic refinement is able to double the size of the modelling. There are several reasons for that: every part of the concrete model is usually broken down into many intermediate levels, which produce many lines of B (hence many lines of target code), and code is not shared through the refinement process (for example, generic elements share a similar code, however the code is duplicated at each use). On the proof obligations side, Meteor, Canarsie and Val Roissy generate respectively around 27 000, 82 000 and 43 000 proof obligations. Even if it is difficult to compare proof just from these figures, we can notice automatic refinement generates more proof obligations. Most of them are located in the abstract model, demonstrating the effectiveness of automatic refinement. Moreover, while 1 400 mathematical rules

[5] Safety Integrity Level 4 (highest level)

were required to prove Meteor proof obligations, only 290 rules were required for Val Roissy (together with 61 generic demonstrations, 97% proof obligations were automatically demonstrated).

Table 1. Number of lines of B models per project.

Project	Abstract model	Concrete manual	Concrete automatic
Meteor	<-	115 000 ->	None
Canarsie line	125 000	38 000	110 000
VAL Roissy	38 000	27 000	117 000

BART has been used for the development of several SIL4 T2[6] tools, mainly to avoid the development of two redundant software (B allows to develop single SIL4 proven software, non-proven software requires to be developed twice by two separate teams with distinct technologies). Figures similar to edithB have been measured from these developments:

- manual modelling represents one third of the total while automatic refinement is two third. Around 500 refinement rules have been added in every project.
- abstract modelling represents two third of development + proof cost while concrete modelling is one third. However 1 800 mathematical rules were added to the prover (compared to the 290 for Val Roissy). This can be explained by the fact that Atelier B main prover mathematical rules database has been enriched to cover the same kind of modelling (ATP software), so the new kind of modelling brought by these T2 tools requires to slightly bootstrap the prover.
- generated source code has twice the size of handwritten source code because of the many operation calls. Because of constraint [***R4***], every refinement step requires to add an extra-layer of imports, hence the number of lines.

Generated models keep track of the initial substitution and of the refinement rules used, with comments inserted in the model (see listing below).

```
Load_component_2(pcd, pid, pctd2) =
  VAR
    l_1
  IN
    l_1 :(l_1 : INT);
    /* Rule : specific.r14_exit */
    /*? l_1 := perm_component_cpt ?*/
    /* Rule : default.default */
    l_1 <-- Load_component_2_1 ;
    test_max_component_def(l_1) ;
```

[6] According to EN50128 standard, tools are categorized in three : T3 for tools directly contributing to source code, T2 for tools performing verification, T1 for all other tools (editors for example).

```
	/*? perm_component_r( bijection_component_def (perm_component_cpt
+ 1)) := pid ?*/
	/* Rule : default.default */
	Load_component_2_2(pid) ;
	…
```

Refinement rules are mainly gathered in a single file (rmf) shared in the whole project. This file has to be set up right at the beginning of the project by identifying the abstract data types that need to be taken into account and how they are implemented. This identification requires performing several interactive refinement sessions (see Figure 9). Then automatically refining the project is initiated, the components being refined in parallel by different users. New refinement rule is added in the shared rmf file if the rule is generic (reusable in another operation / component) or added to a rmf file specific otherwise. Shared rmf file is composed of:

- 400 rules for operations
- 45 rules for structure
- 30 rules for data
- 30 rules for initialisation

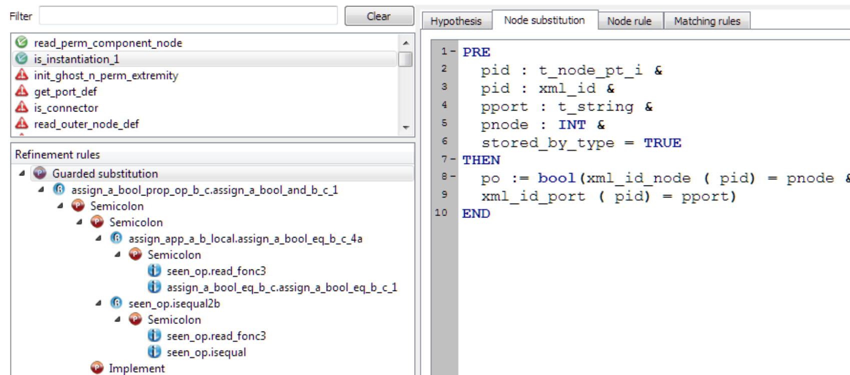


Figure 9: example of interactive refinement session. The refinement tree is displayed on bottom left

5 Return of Experience on Using Set Theory Modelling

The approach presented in this paper is aimed at developing abstract deterministic software specification, using B as a high level language, and leaving most of implementation details to a mechanical, semi-automated process. The specification has to be written using set theory and first order logic, in a way to avoid introducing programmatic gimmicks too early.

For example, in the case of the block controller in [2], blocks are either occupied by a train or free. This information should be safely computed by the functional module “Block Logic”. It is formalized by the abstract variable occupied_blocks:

- Either as a subset of t_block (1): a block belongs to occupied_blocks if and only if it is considered to be occupied.
- Or as a total function from t_block to BOOL (2), associating to a block the value TRUE when the block is occupied and FALSE when it is not.

(1) $occupied_blocks \subseteq t_block$ (2) $occupied_blocks \in t_block \rightarrow \mathrm{BOOL}$

Actually this second choice suits less the proposed B Method, since it is more an encoding of the former (i.e., the use of a set characteristic function instead of the set directly). In this case using a subset of blocks is more abstract. Expressions, predicates and substitutions concerning this set are also more abstract and closest from informal specification. We prefer to use properties over sets when specifying instead of describing the intricate loops that are required to iterate over tables.
We also use relations, partial functions and total functions to type abstract data. For example, the abstract constant ctx_next_block_up associates to a block its next upward block. A block has at most one next block located in the upward direction. A terminal block in the upward direction has no next upward block. So next_block_up is a partial function from t_block to t_block. Finally property states are added to the model, such as this one (linking blocks states and sensors):

$$\begin{aligned}&\forall block \cdot (block \in t_block \wedge \\&\quad ((ctx_block_bs_up[\{block\}] \cup ctx_block_bs_down[\{block\}]) \cap cut_beam_sensors \neq \varnothing \vee \\&\quad ctx_block_detector[\{block\}] \subseteq occupied_block_detectors) \\&\quad \Rightarrow \\&\quad block \in occupied_blocks)\end{aligned}$$

With such predicate, we are able to capture a property in a compact form that is quite straightforward to check. Up to 16 properties have to be formalized in the abstract model. Although it seems to be a small number, some properties had to be cut into many actual properties. At the end, the size of the static and dynamic properties in the main abstract machine is more than 1,000 line long. With the mathematical proof, we do not have to verify the consistancy of these 1,000 lines but just to check that they correctly refine a unique property that is easier to check against the informal specification. In the case of xml engineering tools, xml models are represented as lists, lists of lists, etc. Data types are more elaborated with the notion of node pointers and references to be able to navigate the xml trees. Abstract iterators operate on (node) lists, so many of the existing refinement rules are still valid in this context.
However the development of software aimed at a very different domain induced important modelling effort:

- New data types and structures imply to completely revisit data type related refinement rules
- Existing structural refinement rules have to be specialized in order to take into account these new data types and structures

From this experience, we may consider that the first application of automatic refinement to a new domain is not expected to be economically sound as some effort is required to axiomatize refinement for this domain. However further applications

allow to reuse most of existing rules and to specialize only when required, for particular algorithms. Similarly automatic refinement was initially expected to help newcomers to apply refinement more efficiently. It appears that the profile required to efficiently use automatic refinement demands so many skills that finally only proficient practitioners and experts are able to deal with it. Indeed writing refinement rules could be seen as abstracting the refinement process and requires a strong habit of it.

5 Conclusion and perspectives

Automatic refinement is likely to improve productivity by automating tasks, leading to simpler proofs and simplifying the reuse of known refinement patterns. Similar types and treatments are always refined the same way. This process is not limited to a single type of application, as it has been applied to embedded software as well as to more classical applications (symbolic computation, SysML model analyser / transformer), while obtaining the same metrics (modelling, refinement and proof effort). The process is scalable.

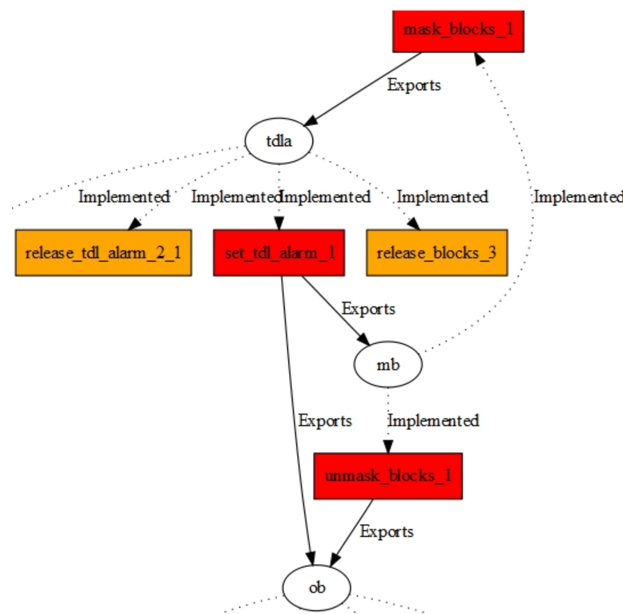


Figure 10: conflict depicted to the user

On the other hand, even if it was our objective when we decided to develop BART, an automatic refiner is not a tool for beginners as it requires understanding why the automatic refinement process stops and how to modify existing refinement rules. Having some experience in refining in B is mandatory to get the most out of it.

Constraint [***R1***] is posing a problem to existing refinement algorithms: in case of conflicting rules, a single variable could be implemented in different components at the same time that is rejected by Atelier B project checker. We are currently working on improving these algorithms in order to not apply conflicting rules and to provide some guidance to the user (interaction with user still remains to be decided). Current feedback is provided as an implementation graph exhibiting *exports* and *implemented* constraints. In Figure 10, we can clearly see a cycle (the upward arrow from variable *mb* to component *mask_blocks_1*). The art of the automatic refinement is today to understand from this graph how to modify refinement rules in order to solve these conflicts.

We are also investigating the possibility to "prove" refinement rules in order to consider that related proof obligations are "by construction" already proven. We are using an approach similar to the Atelier B main prover: the inference engine is based on transformation rules that have been formally proven, hence the successive application of these rules to solve a predicate is considered correct and the predicate is considered true.

References

1. Abrial, J.R. (1996) , *The B-book: Assigning programs to meanings,* Cambridge University Press
2. Amelot, A. & al (2005), *Using B as a High Level Programming Language in an Industrial Project: Roissy VAL*, ZB 2005
3. Behm, P. & al (1998), *METEOR: an Industrial Success in Formal Development*, B'98
4. Behm, P. & al (1999), METEOR: a Successful Application of B in a Large Project, FM'99
5. Benveniste, M. & al (2009), *A Proved "Correct by Construction" Realistic Digital Circuit*, RIAB, FMWeek 2009
6. Burdy, L.(1996) *, Automatic Refinement. In Proceedings of BUGM* at FM'99
7. Casset, L.*,(1999) A formal specification of the Java byte code verifier using the B method,* Lisbonne 99
8. ClearSy (2014), *Atelier B: B language Reference Manual*
9. ClearSy (2014), *BART: User Manual*
10. Essamé, D. & al (2007), *B in Large-Scale Projects: the Canarsie Line CBTC Experience*, B 2007
11. Hoffmann, S. & al (2007), *The B Method for the Construction of Micro-Kernel Based Systems*, ZB 2007
12. Lecomte, T. (2008), *Safe and Reliable Metro Platform Screen Doors Control/Command Systems*, FM 2008
13. Lecomte, T. & al (2007), *Formal Methods in Safety Critical Railway Systems*, SBMF 2007
14. Requet, A. & al (2008), *BART: A Tool for Automatic Refinement*, ABZ 2008
15. Sabatier, D. & al (2006), *Use of the Formal B Method for a SIL3 System Landing Door Commands for line 13 of the Paris subway*, Lambda Mu 15
16. Sabatier, D. & al (2008), *FDIR Strategy Validation with the B method*, DASIA 2008