Automatic Refinement

Lilian Burdy[1,2], Jean-Marc Meynadier[1]

[1] Matra Transport International
48-56 rue Barbès 92120 Montrouge
tel : 01 49 65 71 41 fax : 01 49 65 70 55
e-mail : {burdy,meynadier}@matra-transport.fr
[2] Laboratoire CEDRIC
Conservatoire National des Arts et Métiers
292 rue St Martin 75141 Paris Cedex 3
e-mail : burdy@cnam.fr

Keywords: Tools, Refinement Techniques

1 Introduction

B is a method for specifying, designing and coding software systems. It ensures, thanks to refinement steps and mathematical proofs, that the code satisfies its specification. Up to now, designing and coding are manual and costly activities. We could reduce cost drastically by automating these activities, in other words by automating refinement steps. This paper presents our work on this this automation.

In section 2 we describe the design and coding phase and the gain we can hope from automating these activities. Then in section 3 we present briefly the first specification of the tool we made: the *automatic refiner tool* is considered as a classical compiler. The results of the application of a prototype on two industrial projects and the questions and problem it raises are also described. In section 4, we try to answer these questions by a new specification: the *refiner* becomes similar to a *prover*: the refinement schemes are described by rules, and the tool works automatically or interactively using refinement rules. In section 5, we give an example of refinement rules and the result of its application on an abstract machine.

2 Design and Coding Phase with B-Method

2.1 Background

There are different ways to use the B-Method. Figure 1 gives the process we have chosen and compares it with a classical non formal development.

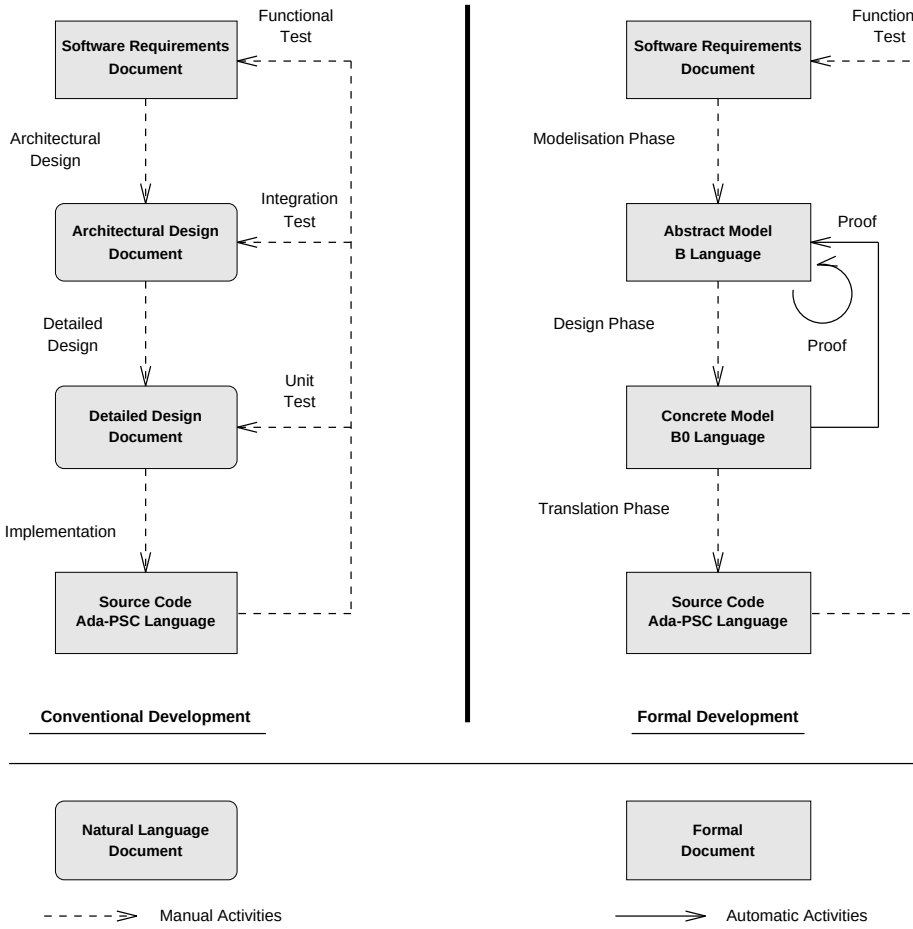

Fig. 1. B vs Classical life cycle.

Modelisation First of all, the modelisation phase consists in translating the software requirement in an abstract model in B (the B specification). This model should:

- describe the software requirement completely, as clearly as possible (it means in particular with mathematical data type) and with a very good traceability,
- let the design work as free as possible,
- allow an easy proof work.

This model is finished when all the requirements are described in the model.

Designing and Coding The following phase consists in refining the abstract model into a concrete model. The model should:

- refine all the components of the abstract model,
- take into account the design constraints in particular in term of memory and speed,
- ease the proof burden.

This model is finished when all the components of the abstract model are refined into components that can be automatically translated into Ada code (it means that these components are in B0, a subset of B defined in the B-Book and very close to Ada code).

Automatic Translation into an Imperative Language A tool translates automatically all the implementations of the concrete model into an imperative language (Ada code in our case).

Consequently, an automatic refiner would cover the design and coding phase, and we could generate automatically the Ada code from the B specification (the abstract model).

2.2 Expected Savings

Referring to the figure 1, and according to our experience on industrial projects, the division of the costs between the different products is the following:

- *abstract model*: 3/8,
- *concrete model*: 3/8,
- *proof of abstract and concrete model*: 1/8,
- *documentation of abstract and concrete model*: 1/8.

Moreover, the main part of proof cost concerns the concrete model, and globally, the design phase needs a bit more than a half of the formal development. *Consequently, an automatic refiner could allow to save more than half of the cost of a formal development.*

2.3 More Details on Refinement

In the B-Book, refinement is presented as conducted in three different ways:

- the transformation of mathematical data structures (sets, relations, functions, ...) into the classical data structures of programming (simple variables, arrays, ...).
- the removal of the non-executable elements of the pseudo-code (pre-condition and choice),
- the introduction of the classical control structures of programming (sequencing, loop).

In order to explain how the refiner tool works, we present the refinement with a slightly different approach:

- *data refinement* is the refinement of variables and the refinement of the without-structure substitutions (*assignment, becomes such that, becomes a member of*). We consider that a refinement of a variable depends on the different write and read non-structured operations where this variable occurs.
- *structure refinement* is the refinement of the structured substitutions (*SELECT, IF, CHOICE, ANY* [1]). It can be done independently of the data refinement and it is well differentiated.

These notions will be used in the next section.

3 Automatic Refiner: a Compiler

Our aim is to build an automatic refiner which can refine a source with abstract B (a part of the abstract model) in B0. By a source with abstract B, we mean a machine with not directly implementable data such as sets, relations or partial functions and non-implementable operations such as adding an element to a set or intersecting two sets.
In this section, we present a general idea of the specification of this tool and also the limitations of a refiner seen as a classical compiler.

3.1 Principles

The abstract model may contain non-deterministic substitutions. We have chosen to bypass this problem. We consider that the entries of our compiler are machines with operations which are completely deterministic.
We do not present a complete specification of this tool; we only present the principles that have guided us:

[1] We consider *ANY* as a structured substitution since we refine *ANY x WHERE P THEN S END* by *x : (P);S* which corresponds to two non-structured sequenced substitutions.

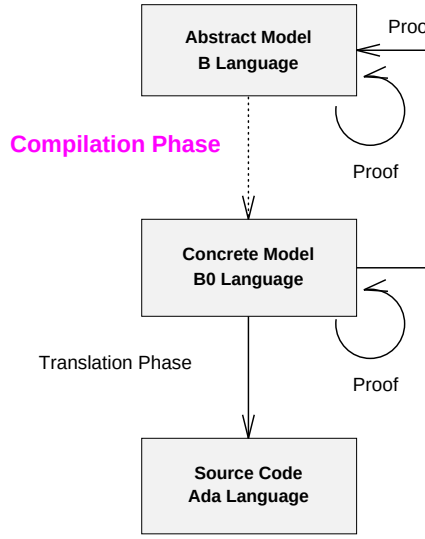

Fig. 2. Refinement as Compilation

- From a machine, it should construct an implementation and if necessary an imported machine with the specification of called operations.
- It is called on each created machine until it creates an implementation without called operations.
- The *structure refinement* is done in a distinguished first step.
- Then, the *data refinement* begins. If the tool does not know how to refine a data or an operation on this data, the data and the operation which concern it are deported in a machine which will be refined by a developer.
- The usual refinement proof obligations are still produced and proved, so that we do not have to validate the tool.

3.2 Prototype

A tool was developed and tested on already existing software and for new software of an industrial project.

Tool applied on already existing software : the size of this software was about 25.000 B lines giving 19.000 Ada lines. Thanks to the tool, with a limited number of refinement schemes, we have produced automatically from the abstract model 75% of the concrete model. The cost of production of this concrete model was drastically reduced compared to the effective cost that had been measured when the concrete model had been manually developed. Moreover the performance of this concrete model was rather good: we had lost only 15% in execution time compared to the manual code, and the size of the executable code did not inflate.

Tool applied on new software : we have had the same kind of results concerning cost effectiveness: the tool produces entirely the concrete model and allows us to make in one week, what was planned to be done manually in more than 2 months.

3.3 The Limits of a Refiner which would be a Compiler

Nevertheless, the automation of the refinement of the abstract model raises a few questions that were not dealt with by this tool:

- in a classical language, we can adopt one good way to "implement" the primitives constructs of the language, but in B, seen as an abstract language, there are many ways to implement a specification and for a given specification there are many possible refinements. None of them can be considered as the best: it depends on the software constraints. For example, one refinement may save memory but costs time, whereas another may save time but costs memory; a third maybe in the middle! How to automatically find them and chose among them ?
- the input language is very large: indeed the specifications may use various data structure, and all these data structure may combine with various control structure. Then, shall we reduce the input language of the tool ?

Hence, we know that we cannot have a refiner which, in any case, refines with the same refinement scheme the same type of data or the same structure. It would give ineffective code most of the time and it would need to be changed to recognise new schemes. So our first refiner was a good compiler for our application since it was purpose built to refine it, but if it is used to refine others sources, it could be really inefficient. For this reason why we have opted for an automatic refiner with a *refinement rule-base*.

4 Automatic Refiner with Refinement Rule Base

The idea was to have an automatic refiner which refines a machine by applying different refinement rules. If these rules are changed, then the produced refinement is different. The refinement is completely parameterised by a rule base of refinement schemes, the refinement of a machine being dependent of this rule base. The rules are completely reusable to refine other machines and the advantage is that the rule base can be modified without modifying the automatic refiner.

4.1 Automatic Refinement

We have kept the idea of the previous chapter about the distinction between structure and data refinement. But this new tool, is divided into two parts:

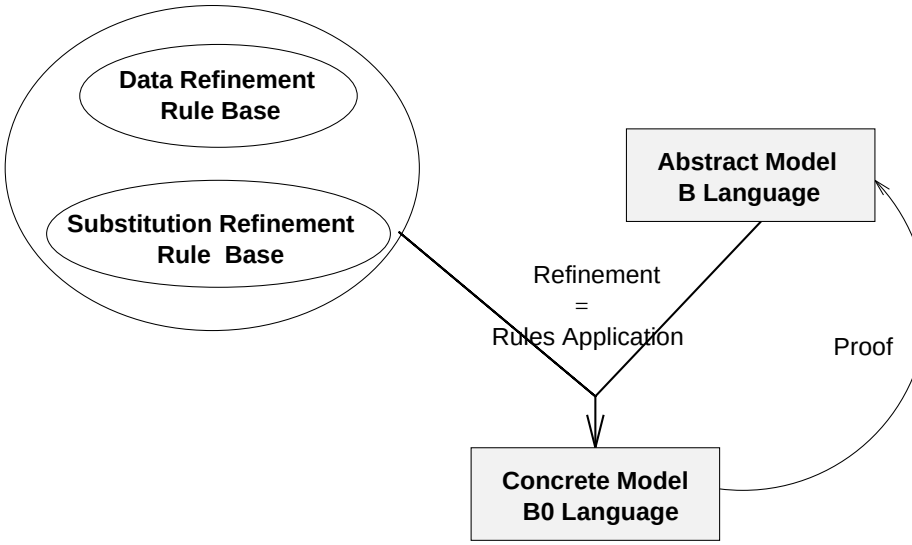

Fig. 3. Refinement as Rule Application

- first part (*structure refinement*): it refines fully automatically the structured operations and produces implementations (with structured operations) and machines without structured operations,
- second part (*data refinement*): it refines automatically or semi-automatically the machines without structured operations (produced by the first part) and produces all the remaining implementations.

The first part of the tool is very simple. We will focus on the second part of the tool. For this second part the activity of refinement can be divided into two sub-activities : variable refinement (and the variable refinement rules) and substitution refinement (and substitution refinement rules).

Variable Refinement Refining an abstract variable is defining the concrete variables that refine it, and for each variable a translatable type and gluing the concrete variables with the refined variable. The choice of such a refinement is done from the abstract type of the variable, the properties of the variable and the operations that are done on it. For example a set is usually refined by a boolean array but if we only access its maximum, it can be refined by a scalar.

Substitution Refinement To refine a non structured operation, we refine each substitution of the operation. The refinement applied to the variables that appear in the substitutions should be known. Then, according to those refinements, one can write the refinement of the substitution. The refinement of substitution can be a non-translatable substitution, in this case this new substitution is itself refined until a translatable refinement is reached.

4.2 Principle

In order to simplify the tool, we have chosen to refine the variables without taking into account the substitutions on those variables. The inconvenience is that the refinement of a variable only depends on its property and must be valid for every conceivable operations and so it must be the most general ; but we consider that a particular refinement can be applied to a variable with, for example, a special pragma that indicates which rules of refinement are to be used on a particular variable.
Then, when the choice of a data refinement is done for each variable, one refines the operations according to these refinements. Those different steps are described in the figure below.

```
MACHINE
 mch
SEES                    } Extract hypothesis from the
 mch_1                  } INVARIANT, PROPERTIES and
 mch_2                  } ASSERTIONS of seen machines.
ABSTRACT_VARIABLES
 x
                        } Associate to each variable a data refinement
INVARIANT               } rule and complete the hypothesis with the
 x ⊆ t                  } information of variable typing.
                        }
INITIALISATION          } Refine the initialisation via the search of a rule
 x := {}                } in the refinement rule base which matches with
                        } the context until a B0 substitution is obtained.
OPERATIONS              }
 op_1 = ...;            } Refine each operation via the search of a rule
 op_2 = ...;            } in the refinement rule base which matches with
 ...                    } the context until a B0 substitution is obtained.
END                     }
```

4.3 A Refiner is like a Prover

To understand how the refiner works, it can be compared with a prover. In fact, we have got some hypothesis and a goal which is the variable or the substitution to refine. One tries to apply rules which are presented as a refinement general schemes. These schemes can be applied on some substitutions under some hypothesis. When the rule to apply is chosen, one obtains a new substitution that is terminal if translatable or on which another rule should be applied otherwise. This process is repeated until the substitution is discharged e.g. completely refined.

RULE R1 IS
ABSTRACT_VARIABLES
a
TYPE
ENS(a,b,c)
CONSTRAINT
(SETS(c) ∧
b ⊆ c ∧
a ⊆ b) ∨
(SETS(c) ∧
a ⊆ c ∧
match(b,c))
CONCRETE_VARIABLES
a_r
INVARIANT
a_r ∈ c → BOOL &
a = a_r^{-1}[{TRUE}]
END

Fig. 4. The set data refinement rule

4.4 Example of Refinement Rules

Data Refinement Rule The rule of figure 4 is a data refinement rule that is applied on a variable a if one sub-constraint of the disjunctive constraint is solved. The clauses are instantiated and then, the information concerning the refinement are written in the implementation : the concrete variable, its type and the gluing invariant. The instantiated predicate $ENS(a, b, c)$ becomes a new hypothesis, which will be useful to refine the substitution where this variable occurs.

Substitution Refinement Rule The rule of figure 5 is a substitution refinement rule that is applied on a substitution $a := a \cup \{b\}$ if a has been refined with the previous rule e.g. if a predicate matching with $ENS(a, c, d)$ is in hypothesis. If these conditions are fulfilled, the refinement is applied. This refinement is a VAR IN with a sequence in the body. The left part of the sequence is not B0-valid (since b can match with an abstract expression) and so a new refinement step will be done on this substitution. The right part is B0-valid and so, the refinement will stop. Then a translatable refinement is obtained and the implementation can be written.

4.5 The Tool

We have prototyped a tool with this specification. This tool has three inputs :

- a B component (machine or refinement) the user wants to refine,
- a generic refinement-rule data base,
- a set of specific refinement-rules.

```
RULE R2 IS
    REFINE
        a := a ∪ {b}
    CONSTRAINT
        ENS(a,c,d)
    IMPLEMENTATION
        VAR
          l_1
        IN
          l_1 := b;
          a_r(l_1) := TRUE
        END
END
```

Fig. 5. The add an element to set rule

The generic data base is made, today, of 300 rules that comes with the tool. The set of specific refinement-rules contains, if needed, a few rules not present in the generic data base that are needed to refine the B component. These rules are written by the end-user.
The output of the tool is a set of B components that refines completely the B abstract machine, according to the refinement-rules coming from the generic (and user specific) refinement-rule data base.

5 Example of an Automatic Refinement

Let refine the machine *example*

```
MACHINE
 example
SEES
 set_mch
ABSTRACT_VARIABLES
 a_set
INVARIANT
 a_set ⊆ A
INITIALISATION
 a_set := {}
OPERATIONS
 add(b) =
 PRE
   b ∈ A
 THEN
   a_set := a_set ∪ {b}
 END
END
```

```
MACHINE
 set_mch
SETS
 A
END
```

The first step consists in refining the abstract variable, it gives the first goal to prove:

$$
\begin{array}{l}
SETS(A) \wedge \\
a\_set \subseteq A \\
\Rightarrow \\
Refine(a\_set)
\end{array}
$$

The rule R1 is applied since the constraint ($SETS(c) \wedge a \subseteq c$) can be solved with the hypothesis; so the goal is discharged.
The declarative part of the implementation can be written.

```
IMPLEMENTATION
 example_i
REFINES
 example
SEES
 set_mch
CONCRETE_VARIABLES
 a_set_r
INVARIANT
 a_set_r ∈ A → BOOL &
 a_set = a_set_r^-1[{TRUE}]
```

Then we have a goal for the refinement of the operation. We have new hypothesis: the hypothesis resulting of the refining of the variable and some hypothesis extracted from the header of the operation and the precondition.

$$
\begin{array}{l}
SETS(A) \wedge \\
a\_set \subseteq A \wedge \\
ENS(a\_set, A, A) \wedge \\
In\_parameter(b) \wedge \\
b \in A \\
\Rightarrow \\
Refine(a\_set := a\_set \cup \{b\})
\end{array}
$$

The rule R2 is applied since the goal matches with the REFINE clause and the unique constraint is verified by the hypothesis. So a new goal is obtained after the application of the rule.

$$
\begin{array}{l}
SETS(A) \wedge \\
a\_set \subseteq A \wedge \\
ENS(a\_set, A, A) \wedge \\
In\_parameter(b) \wedge \\
b \in A \\
\Rightarrow \\
Refine \left(\begin{array}{l} VAR \\ \quad l\_1 \\ IN \\ \quad l\_1 := b; \\ \quad a\_r(l\_1) := TRUE \\ END \end{array} \right)
\end{array}
$$

Then this goal is divided in two subgoals, each one corresponding to a part of the sequence :

$$
\begin{array}{l}
SETS(A) \wedge \\
a\_set \subseteq A \wedge \\
ENS(a\_set, A, A) \wedge \\
In\_parameter(b) \wedge \\
b \in A \wedge \\
Local\_variable(l\_1) \\
\Rightarrow \\
Refine(l\_1 := b)
\end{array}
\quad \text{and} \quad
\begin{array}{l}
SETS(A) \wedge \\
a\_set \subseteq A \wedge \\
ENS(a\_set, A, A) \wedge \\
In\_parameter(b) \wedge \\
b \in A \wedge \\
Local\_variable(l\_1) \\
\Rightarrow \\
Refine(a\_set\_r(l\_1) := TRUE)
\end{array}
$$

A rule is applied to each subgoal, since they are both translatable, these rules are terminal; and so the refinement stops and the second part of the implementation can be written.

```
OPERATIONS
  add(b) =
  VAR
    l_1
  IN
    l_1 := b
    a_set_r(l_1) := TRUE
  END
END
```

The tool has completely automatically written this implementation. One can remark that it is not the best implementation that can be written for our machine since the local variable is useless here. But our rule R2 is general and is correct in every case even if b matches with a complex expression.

6 Results and Conclusion

We have made an automatic refiner that is ready to be used for industrial developments in our company. Our formal design experience, based on a large formal

development (more than 115.000 B lines), is now largely formalised in the *refinement rule base* (more than 300 refinement rules). This base is completely reusable for the new developments, and new rules coming from new developments may enrich it.

This tool has been used already on industrial developments in our company. The productivity gain is very important. On the last project, thanks to the tool, 20.000 B lines have been produced (corresponding to 14.000 Ada lines) in 5 man days. More than 97% of the refinement lemmas generated have been proved automatically. Moreover, as we know that the machine and implementation produced by the tool are correct (provided that the refinement rules are correct), the interactive proof is more effective.

By now, the developers can focus on the modelisation phase since the design phase is largely automated. We can hope that B will be soon a modelisation language and the design phase will be, as the translation, a push-button activity.