Adaptable Translator of B Specifications to Embedded C Programs ⋆

Didier Bert[3], Sylvain Boulmé[3], Marie-Laure Potet[3], Antoine Requet[2], and Laurent Voisin[1]

[1] ClearSy, Aix-en-Provence, France
Laurent.Voisin@clearsy.com
[2] Gemplus Research Labs, La Ciotat, France
Antoine.Requet@gemplus.com
[3] Laboratoire Logiciels, Systèmes, Réseaux - LSR-IMAG - Grenoble, France
{Didier.Bert, Sylvain Boulme, Marie-Laure.Potet}@imag.fr

Abstract. This paper presents the results of the RNTL BOM project, which aimed to develop an approach to generate efficient code from B formal developments. The target domain is smart card applications, in which memory and code size is an important factor. The results detailed in this paper are a new architecture of the translation process, a way to adapt the B_0 language in order to include types of the target language and a set of validated optimizations. An assessment of the proposed approach is given through a case study, relative to the development of a Java Card Virtual Machine environment.
Keywords. Code generation, embedded systems, B method, smart cards.

1 Introduction

Formal methods aim to produce zero-defect software, by controlling the whole software development process, from specifications to implementations. In top-down approaches, they start from high-level and abstract specifications, by describing the fundamental properties of the final product. Details and design choices are introduced in an incremental way. The correctness between two levels is insured by refinement proofs. When implementations are aimed, refinement leads to a last level which describes, in some way, the expected behavior. Implementations are generally stated in an appropriate sub-language of the specification language. The code generation process consists in two stages: formal implementations are translated into programs in a given programming language, and then these programs are compiled. This approach offers several advantages: the translation process is as simple as possible, and it can be validated in an easy way; secondly appropriate compilers can be used, as optimizing and/or certified compilers. The simplicity of the translation ensures traceability between the formal specification and the executed code.

⋆ This work has been supported by the French Ministry of Industry in the RNTL program (Réseau National des Technologies Logicielles), under project BOM (B with Optimized Memory), March 2001-March 2003.

Nevertheless all approaches which support formal development from specification to code must manage several constraining requirements, particularly in the domain of embedded software where specific properties on the code are expected, such as efficiency or memory size. First a compromise must be found between the expressiveness of the formal implementation language and the simplicity of the translation process. Another compromise is also necessary between the formal models, which generally favor the readability and the simplicity of the verification process, over the code efficiency. Finally it is not possible to define a unique translation process, which is proved adapted for all uses. A translation process is generally redefined depending on the range of products or target platforms.

The RNTL BOM project is supported by the French Ministry of Industry and involves two academic laboratories and two industrial partners: Gemplus who uses the B method for smart card applications and ClearSy who provides Atelier B. The aim was to develop a new translation approach to generate efficient code from B formal developments. The B method has been successfully applied for industrial projects, particularly in the domain of railway-automated systems [2]. For such applications adequate translators have been developed, thanks to the technique of the Vital Coded Processor which secures the execution process. In the domain of smart cards, Gemplus has used the B method for certification purposes [15, 16] or to develop correct code [8]. When implementations are expected, the generated code must meet the card's requirements. In particular the memory size is limited. For such applications, the Atelier B tool is not well-suited. Among others, the needs were to determine:

- soundness conditions to optimize parameter passing mechanisms and to perform operation inlining,
- a mechanism to integrate some basic C types in the last level of refinement.
- a new architecture of the translation process to easily adapt it with respect to target platforms.

The first requirement aims to optimize the memory size and to eliminate operation calls, introduced by the modeling process. The second requirement aims to give freedom to the developers to adjust at best their integer representation. The last one is crucial for the smart card domain where applications must be loaded on different platforms, with their own characteristics.

This paper describes how these requirements have been incorporated in the B method. Section 2 introduces some features of the B method which are important for the translation process. Conditions to optimize parameter passing are also presented. Section 3 presents the architecture and the various parts of the translation process. Section 4 explains how adaptability can be obtained at the level of the B_0 language and at the level of the translation process. Finally, Section 5 presents the result of a case study, relative to the development of a Java Card Virtual Machine environment. The code is compared with other translators and the impact of the optimization is assessed.

2 Some B Features

2.1 B methodology

The B method supports a formal development approach in a top-down manner. In particular, data may be refined. That is expressed using a gluing invariant, linking "internal" data to "observed" data. Proof obligations guarantee that observable results are consistent with the higher levels of the specification. The last step of refinement is the implementation itself, written in a programming language called B_0. Moreover modular developments are also supported by the B method. Specifications can be composed and then refined separately.

The B_0 programming language is designed as a relatively low-level language. This choice has two advantages: it allows the developer to control the generated code (the code eventually executed), and it provides safety. Indeed, the job of the translator is simple and the final generated code uses only finite memory. In B_0, data-types (integers) are bound, there is no dynamic allocation, and no recursive call. Thus, it is theoretically (currently no tool performs this job) possible to bound statically the memory size needed by a B_0 program. On a checked development, the B method guarantees the total correctness of the development. It means that programs terminate and no runtime errors will occur on the generated code (if there is sufficient memory to run the program). More precisely:

- B_0 type-checking guarantees usual programming typing properties: right-hand side and left-hand side of an assignment get the same type and the type of actual parameters (input and output) is the same as the formal parameter ones.
- Proof obligations of "*well-definedness*" [3, 6] guarantee that partial operators are soundly applied (arithmetic overflow, division by zero, access to array elements, etc.). That is verified by the proof that, in a call $f(e)$, the arguments e belong to the domain of the function f and that f and e are also well defined.
- *Well-definedness* guarantees also that values of variables always inhabit their declared type (or in other words, that variables are initialized before they are read).

2.2 Modularity of B specifications

Specifications of large applications cannot be carried out in one block. The B language provides primitives to specify pieces of a problem, to develop them independently, and to compose them in such a way that the properties proved in a local part are preserved in the global development after composition.

High level specifications are expressed in MACHINE components. They declare the safety properties, which must be preserved in the final programs, the interface of the component (list of operations) and the specification of the internal state and of the operations. Machines are refined into REFINEMENT components and

then in IMPLEMENTATION components which constitute the programming level. Machines or refinements can be built incrementally by including or by using other machines (clauses INCLUDES and USES).

Implementations can call operations of other machines which are imported (clause IMPORTS). This corresponds to a decomposition in layers. Finally a component can see another machine to share its services with other components (clause SEES). The opposite figure shows a typical B development. Each clause introduces specific syntactic restrictions to compose safely invariant and refinement proofs [5, 19] (in particular variable sharing is strictly mastered).

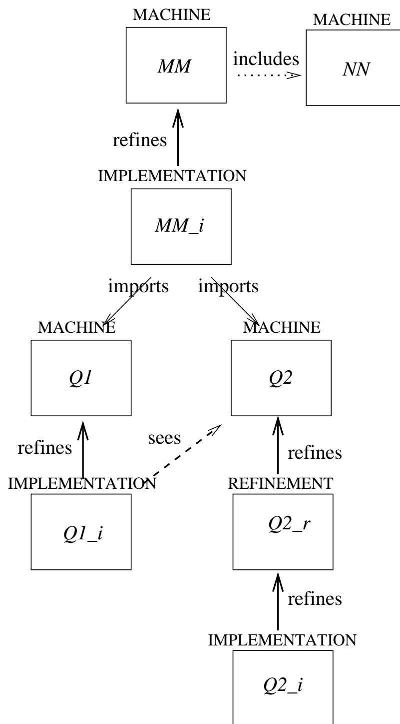

Fig. 1. A modular B development

Each component encapsulates an internal state made of variables, specified by a predicate called *invariant*. The link between the states of components connected by a refinement relation is performed by a *gluing invariant*. This invariant binds the variables at the higher level to the variables at the lower level. A particular case of binding consists in giving the same name to a variable in two adjacent levels. The gluing invariant is then the equality of the values of both variables. This principle of name equality is also allowed between two development chains. For instance, in the implementation *MM_i*, one can glue a name coming from *MM* with the same name in machine *Q2*. This mechanism is called *gluing by homonymy* and provides a facility to develop simpler models.

At the leaves of a B project, one find either implementations, which can be translated directly, or *basic machines* the implementation of which is carried out in the target language. Those latter machines thus constitute the interface between B and non B parts of a project.

2.3 Abstract and concrete data

In the B language, there are two kinds of constants or variables (*data*): abstract data and concrete data. Abstract data consist in all the elements and sets that can be used in set theory, as defined in full B (integers, enumerated sets, Cartesian products, power sets, relations, and so on). They are mainly used at the higher levels of specification (machines and first refinements). Concrete data are those which may be used in B_0, each time the data thus introduced will not be further refined in the development. It is the case for constants or variables at the

implementation level but also for parameters and results of operations, which cannot be refined in B. Concrete data must match ordinary data types in usual programming language because they should be "implemented" directly in the target language. So, the correspondence between concrete data and data types must be obvious. In the sequel concrete data types will be called "B_0 types". In standard B, they are the following ones:

- enumerated types (including the boolean type)
- bounded integer type (from `MININT` to `MAXINT`)
- arrays on finite index intervals where the type of elements is a concrete type (in set theory, they are similar to total functions)
- deferred types (types without representation). They will be represented by definite concrete types at the end of developments.

2.4 Abstract and concrete arithmetic

In the B method, if a variable is declared by $x \in \mathsf{INT}$ (the predefined bounded integer set), it means that its value is an integer value in the mathematical set $\mathbb{Z}$ and it satisfies the property $x \in \mathtt{MININT} .. \mathtt{MAXINT}$. An expression as "$x+1$" is well defined in the abstract levels, because 1 is a constant value in $\mathbb{Z}$ and $+$ is a mathematical operation defined in $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$. In the implementation level, the interpretation is not the same. The denotation INT becomes a B_0 type similar to a type in programming languages. The predefined operators $+$, $-$, etc. in the implementations are actually operations on this type, bounded by $\mathtt{MININT} .. \mathtt{MAXINT}$. Concrete and abstract operators are overloaded in the B language, but we shall denote the concrete ones by $+_0$, $-_0$, etc. to avoid confusion. Thus, the "$+_0$" operator is defined by[4]:

$$+_0 \mathrel{\widehat{=}} \lambda(x, y) \cdot (\ x \in \mathsf{INT} \ \wedge \ y \in \mathsf{INT} \ \wedge \ x + y \in \mathsf{INT} \mid x + y \ )$$

Clearly, this definition contains a new restriction with respect to the mathematical $+$ operation, that is the operands and the result must satisfy the interval constraint. Thus, when refining abstract integers with concrete ones, *well-definedness* proof obligations ensure that no overflow will happen, even in intermediary results.

2.5 Parameter passing

Parameter passing mechanisms are often the stumbling block of the semantics of programming languages. It is one of the crucial points for the correctness of program translation or compiling [14]. In this section, we show how the B theoretical parameter passing of the operations can be soundly translated into the classical parameter passing mechanisms of programming language. In the B language, operations are declared at the level of machines by a text of the form:

[4] In notation B, $\lambda x \cdot (\ P \mid E\ )$ is a function where the value is E iff parameter x satisfies predicate P (preconditions).

$$r \longleftarrow op\,(p) \mathrel{\widehat{=}} \text{PRE } P \text{ THEN } S \text{ END}$$

where p is a list of formal input parameters and r is a list of formal output parameters. Predicate P is the precondition and S is the body of the operation. A call to the operation op is written as $v \longleftarrow op\,(e)$ where v is the list of actual output parameters (a variable list) and e is the list of actual input parameters (an expression list). The semantics of the operation call is defined in the B-Book [1], by the following substitution rule, which means that r and p are respectively replaced by v and e in P and S:

$$v \longleftarrow op\,(e) \;\equiv\; [r, p := v, e]\,\text{PRE } P \text{ THEN } S \text{ END} \tag{1}$$

The syntactic restrictions of the call imply that v is a list of single variables without repetition (not array elements, for example) and that these variables do not occur in P and S (non aliasing). Definition (1) is well suitable for proofs, because proofs rely upon the specification of the operations where the state modification is atomic (i.e. specified in one step). But this rule is not an operational one because an operation call must indeed be translated by a call to the implementation of this operation. Rule (1) cannot be directly applied to the implementation, because operation bodies may contain sequencing of substitutions and while loops. So, it must be changed in the following equivalent rule, which can be applied to the implementation level:

$$v \longleftarrow op\,(e) \;\equiv\; \text{PRE } [p := e]\,P \text{ THEN VAR } p, r \text{ IN } p := e; S; v := r \text{ END END} \tag{2}$$

This rule is usually identified as *by copy* parameter passing. Here again, syntactic restrictions on variables allow deducing the new parameter passing rule:

$$v \longleftarrow op\,(e) \;\equiv\; \text{PRE } [p := e]\,P \text{ THEN VAR } p \text{ IN } p := e; [r := v]\,S \text{ END END} \tag{3}$$

The equivalence between (2) and (3) relies upon the fact that variables in list v do not occur in S. So, actual output parameters can be substituted into S, which is called *by reference* parameter passing. This rule is used by the regular translators of Atelier B. Moreover, it can be proved that input parameters can also be passed by reference, under the following condition:

Condition 1: [By-reference parameter passing condition] Actual input parameters do not contain any occurrence of actual output parameters, nor any occurrence of variables that the operation works with.

Under Condition 1, rule (1) can be applied to the bodies, also in the implementations. The proof of this condition is given in [18]. So, parameter passing in operation call in B can be done by reference if Condition 1 is satisfied. This is particularly useful in case of array parameters to avoid a copy of the value.

3 Translator Architecture

The translation process consists in transforming the B_0 part of a B project into a text written in C such that both texts (in B_0 and in C) are observationally equivalent with respect to their respective semantics. To make B translator reusable

for several targets, the translator is split into different parts as shown on Figure 2. In this architecture, tools *Flattener* and *Optimizer* are target-independent. The *Translator* tool must be customized for each new target language or for specific execution platforms.

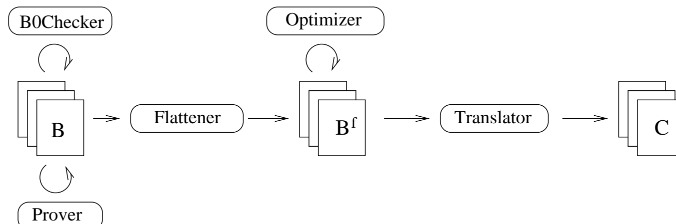

Fig. 2. Translator architecture

The *B0Checker* checks B_0 syntactical restrictions to fulfill the validity constraints determined for BOM translators. The *Flattener* produces compilation units called *flattened modules* (Section 3.1) from structured developments. Moreover, it renames identifiers in order to have a single name space for the whole project (like for instance in C). On the flattened modules, the *Optimizer* performs code transformations preserving the semantics (Section 3.2). At last, the *Translator* produces output files in the target language. For instance, for each flattened module, say *MM*, the BOM translator produces a header file `MM.h` and a code file `MM.c` (Section 3.3).

3.1 Modular flattening of B developments

A *module* is a chain of development starting from a machine and ending with the implementation of this machine, incremented by all the machines which are transitively included and used by the components of this chain. Relations between modules are only relations "sees" and "imports". The name of the root machine becomes the name of the module. A *flattened module* is obtained by gathering all the concrete data contained in a B original module. It contains concrete sets, concrete constants, concrete variables, initialization and operations of the implementation. Concrete data are equipped with their B_0 type. Figure 3 presents the modules of the development of Section 2.2 and the structure of the flattened modules.

To avoid *name clash* between identical names of different modules in a project, the flattener renames with a unique name each identifier of B modules. Renaming is simply performed by prefixing all the names by the name of the module where they appear. This allows one to deal with a single name space for the whole project and makes the identifier generation in the target language very easy: the renamed identifiers are kept as such in C. This provides also traceability: it is easy to know where each renamed identifier is defined.

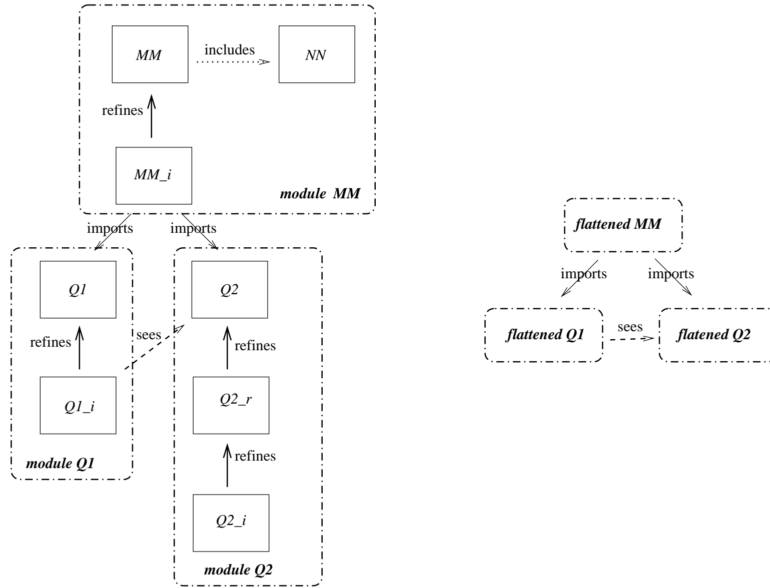

Fig. 3. B modules and flattened modules

However, the B language offers the ability to give the same name in different components to give rise to the same entity by homonymy rule, as explained in Section 2.2. After renaming these names are no more the same. To solve this issue, a flattened module contains lists of pairs of names which have become different by renaming, but actually are the same. For more details, see the B reference manual and the report about renaming in the BOM project [12, 4]. Names which are paired in this list must be eventually represented by the same entity in the target language.

3.2 Global optimization phase

The global optimization phase contains two passes. The first one performs the *inlining* of operations. A pragma can be added to the source specifications in order to indicate to the optimizer which operations must be inlined. Inlining is safe thanks to condition 1 (section 2.5). As illustrated by the case study (section 5), this optimization is very useful, because the B method, as the other top-down methods, can lead to a great number of operations encapsulated in machines. Moreover, the decomposition of a development in small levels enhances the proving power of the tools by making the proofs shorter. By inlining, the number of calls is reduced, thus reducing the call stack and the execution time. Tools could be added at this stage to acquire more information on the effect of inlining and to tune the process according to determined objectives (memory size, execution time or stack size optimization).

The next optimization phase is complementary to the previous one. It consists in removing the operations which are not (or no more) called by the program. This is carried out by a simple examination of the call graph.

The global optimization phase deals with optimizations which are not performed by compilers or which are simpler when treated at this level. It is the case for inlining which is more complicated for a full programming language. Here we exploit the peculiarities of the B_0 language (in particular, there is no aliasing) to obtain a simpler and more efficient result.

3.3 Principles of the translation

Like in a separate compilation process, the translation can be done a module at a time. Hence, the purpose of this step is to translate each flattened module in a *observationally equivalent* standard C module. For instance, an operation op_{B} of a B_0 module is translated in a C function op_{C} with the same name, such that given an instance C_{B} of the *context* of op_{B} and an instance C_{C} observationally equivalent to C_{B}, then op_{C} has a *behavior* observationally equivalent to op_{B}. Let us now make these notions precise.

- the behavior of an operation is characterized by the (abstract) function associating inputs parameters to output and the effect of the operation on the memory state.
- the context of a B_0 operation consists of all the names, visible by this operation, associated with their respective types. An instance of the context consists in associating to each name a value consistent with its declared type. In B_0, there is no aliasing: each state location is referred to by a unique identifier. State locations can be considered equivalent to their associated name.
- the observational equivalence is based on equivalence between B_0 values and C values. This equivalence on values is naturally extended on instances of context.

The observational equivalence between B_0 sets and C types is done as follows:

B_0 types	C types
Enumerated sets	Enumerated types
Basic integer sets	Predefined integer types
B_0 array types	C array types

The links between B_0 and C for integer values has been considered as crucial for the efficiency of the generated code and for the correctness of the translation. So, the solution chosen by the BOM project is to be able to interface very tightly the B_0 integer types and the C integer types. This interfacing is adaptable to various target platforms. More generally, this interfacing is a way to link B_0 types to the types of a given target language. This point is detailed in Section 4.2.

The links between B_0 arrays and C arrays is not straightforward. In B, arrays correspond to total functions whereas in C, they correspond to a contiguous zone of memory (coded as the beginning address of the array and its size). However, it is easy to do a semantical correspondence between an array element $t(i)$ in B and the value at the location `t[i]` in C.

4 Adaptability Means

Adaptability of the translator is achieved by two different techniques. The first one is the possibility to change the target language or to adapt the translation of the B syntactic constructions by the way of translation rules. The translator is thus a single interpreter, which applies the rules when they match the B texts in input, and produces the corresponding text in output. This point is developed in Section 4.1. The second one is the ability to connect a B_0 type with a corresponding type in the target language. Operations on these types are specified in basic machines, which have a direct implementation in the target language. Under the condition that the implementations of basic machines are correct with respect to their specification, then the correctness of the translator is preserved. We detail this mechanism in Section 4.2 on the example of the C integer types.

4.1 Translation mechanism

The starting point of the translation is the language source syntax. To illustrate this point, we give below a piece of the syntax of B_0 module: the definition of an operation. *F_Params* is a non-terminal notion which represents a sequence of formal parameters. We consider that there may be only one result parameter in *Op_Def* for the sake of simplicity. A formal parameter is a pair with the name of the parameter and its B_0 type. Square brackets are put around optional parts in the syntax.

Op_Def ::= [(*Idf*:*Idf*) <-] *Idf* [(*F_Params*)] = *Statement*

The specification of the translator is the description of the links between the programs in the source language and programs in the target language. The translation is *complete* if all the correct[5] source programs can be translated. It is *correct* if the semantics of the translated programs is consistent with the semantics of the source programs. Generally, such a verification is very complex. Here, we are in a particular case because B_0 is very simple and the distance between B_0 and C is short. The translation can be specified for each input syntax rule, what ensures completeness, and the validation can be done inductively on the syntax of the language.

The translation is thus specified using a set of translation rules. Those rules are then interpreted by a tool developed specifically as part of the BOM project. That interpretor is built upon the Logic Solver provided with Atelier B[6]. In fact, the interpretor is a means to tailor the mechanisms provided by the Logic Solver (pattern-matching, term-rewriting, etc.) to the specific needs of translation. The translation rules are expressed by rewrite rules of the following form:

[5] Here "correct" means: syntactically correct, well-typed and proved.
[6] The Logic Solver is a general purpose formula manipulation tool that is used in the heart of the B prover.

operator name (pattern)	heading of the rule
& *list of guards*	optional conditions for applying the rule
=>	separator between antecedent and consequent
description of the result	effect of the rule

A pattern is a text (actually a tree) containing metavariables called *jokers*. For some input fragment, the interpretor tries to match the pattern of the rule with the input fragment, possibly deducing the value of some jokers. It then evaluates the guards. The rule is selected if the pattern matches the input fragment and the guards hold. Otherwise, it is skipped and the interpretor searches for an other rule further in the translator specification. When a rule is selected, its consequent is evaluated, building the output of the translator as a side-effect.

Translation rules thus get a functional recursive form on the structure of the syntax tree. Translation rules may be defined conditionally. For example, an operation definition in B may get a result which is either a scalar value or an array value. In C, functions do not return array values, because array values cannot be assigned to variables. So, in the BOM project, we decided to consider the output array values as input array parameters, but to let the result as a returned value for the output scalar values. Such a condition can be done on the B_0 type of the result parameter. For example, an operation definition with one return parameter and without input parameters is translated by the conditional translation rule below. In that rule, V, T and I are *Idf* variables and $S \in$ *Statement* (jokers). Some operators are introduced to generate strings (e.g. `Write`) or to check conditions (e.g. `Guard`). Function `tr_Proc_Sig(`o`,`p`)` generates the heading of a C procedure with name o and parameters p and function `tr_Func_Sig(`t`,`o`,`p`)` generates the heading of a C function of type t, name o and parameters p. Finally, `f_param` is a constructor, which takes a pair of identifiers (a name and a type) and returns a list reduced to only one formal parameter.

```
tr_Oper( (V:T) <-  I  = S )
& Guard( IsArrayType(T) )
=>
tr_Proc_Sig(I,f_param(V,T)) & WriteLn("{") &
   Indent & tr_Stm(S) & Dedent &
WriteLn("}")
```

```
tr_Oper( (V:T) <-  I  = S )
& Guard( IsScalarType(T) )
=>
tr_Func_Sig(T,I,()) & WriteLn("{") & Indent &
   WriteLn("% %;", T, I) &
   tr_Stm(S) &
   WriteLn("return %;", V) & Dedent &
WriteLn("}")
```

The translator is flexible because the rules are written in one or more files that are interpreted. The classical approach is to describe a standard translation

schema in a main file, and then to refine it by adding some rules in an auxiliary file. The rules in the auxiliary file are fetch first, then in case of failure, the rules of the main file are used. So, auxiliary files allow adaptation of the general translator to specific needs. If a developer wants to adapt the translator, he has to add new rules to the auxiliary file according to the characteristics of the target machine or of the C compiler. Obviously, these new rules must be certified to guarantee the validity of the whole translation process. The list of built-in commands of the translator is determined by the Atelier B provider (ClearSy) and the adaptation of the translator can be done by a developer (e.g. Gemplus).

4.2 Interface with C integer data types

The principle is to associate a B_0 type to each C data type that are considered useful for the programming task. That means that some C data types (at least here the integers) are promoted from the target language into the B language. From a method point of view, if a user wants to generate C code, for example, he has to refine his abstract model towards a model where the integer values are exactly those provided by the C language. So, the correctness between the abstract level and the B_0 level is ensured by the proof obligations of the B method, while the translation between the last B_0 level and the C level is straightforward.

The promotion of the target data types into the B language is realized by *specifying* the former ones in the latter one. For the specification of C types in B, as for the validation of the translation rules, the ISO standard [22] has been followed. For the features which are considered as *implementation-dependent* by the standard, we chose to take the meaning usually adopted by C compilers. In these well-identified cases, a developer of a new translator must check that his own compiler respects the meaning formally specified by the BOM translator. An example of implementation-dependent behaviour is the overflow of an arithmetic operator on signed integers. We illustrate in the sequel the specification of some C integer types.

Following a common use, there exist several sizes for the integer values, each of them being able to contain either signed or unsigned representations. More formally, on a 16-bit architecture, the correspondence between both languages can be described as in the following table:

B_0 type	Formal range	C type attributes
t_int16	$-2^{15}..2^{15}-1$	`int`
t_int32	$-2^{31}..2^{31}-1$	`long int`
t_uint16	$0..2^{16}-1$	`unsigned int`
t_uint32	$0..2^{32}-1$	`unsigned long int`

The new type names are introduced in B by the declaration of concrete constants denoting the associated intervals in basic machines. These types are not compatible and conversions between them must be made explicit by the programmer (on the contrary to C). Several other constants can be added, like `MIN_INT16`, `MAX_INT16`, etc. as it was done for the standard integer type INT (Section 2.4).

Together with the B_0 types, C operations on integers must be specified in the basic machines. In project BOM, two B operations are defined for each C arithmetic operation. The first operation looks like standard B_0 arithmetic operation. It checks (by proof obligations) that the mathematical result is contained in the data type. The second one defines operations with overflow. In that case, a truncation is performed on the result. Notice that if the programmer uses the operation with truncation and if the refinement proofs of the implementation are done, then that means that whatever the result of the arithmetic operation, it is consistent with the abstract level.

For instance, the additions for signed or unsigned integers of size n with and without truncation are specified in B by the declarations:

$$
\begin{aligned}
\mathit{add_int}_n &\mathrel{\widehat{=}} \lambda(x,y)\cdot(\ x \in \mathit{t_int}_n \wedge y \in \mathit{t_int}_n \wedge x+y \in \mathit{t_int}_n \mid x+y\ )\\
\mathit{add_uint}_n &\mathrel{\widehat{=}} \lambda(x,y)\cdot(\ x \in \mathit{t_uint}_n \wedge y \in \mathit{t_uint}_n \wedge x+y \in \mathit{t_uint}_n \mid x+y\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{add_int_trunc}_n &\mathrel{\widehat{=}} \lambda(x,y)\cdot(\ x \in \mathit{t_int}_n \wedge y \in \mathit{t_int}_n\\
&\qquad\qquad \mid ((x+y+2^{n-1}+2^n) \bmod 2^n) - 2^{n-1}\ )\\
\mathit{add_uint_trunc}_n &\mathrel{\widehat{=}} \lambda(x,y)\cdot(\ x \in \mathit{t_uint}_n \wedge y \in \mathit{t_uint}_n \mid (x+y) \bmod 2^n\ )
\end{aligned}
$$

In these definitions, *add_int* and *add_uint_trunc* are the + operator defined in the ISO-C standard, respectively on signed and unsigned integers. Function *add_int_trunc* is the + operator usually implemented by the compilers in case of overflow. Function *add_uint* is provided by the BOM translator as an instance of *add_uint_trunc* in case of non overflow. The semantics of the non-truncating operator produces proof obligations much simpler to deal with.

5 A case study: the Java Card Virtual Machine

5.1 General presentation

In the BOM project, a case study was chosen to assess the new translator of B specifications. The Java Card virtual machine is typically an application which must be embedded in smart cards and which has been intensively studied by Gemplus [20, 11, 13]. This case study completes a formal development of the Java Card byte code verifier using the B method[7] [10, 9, 7]. The subset of the JCVM specified and developed in B is large enough to be able to read the bytecode files of Java applets and to execute them. An application in Java was also written (an electronic purse) to test the embedded virtual machine. A complete JCVM contains four components: loader, linker, Java Card Runtime Environment (JCRE) and interpreter of bytecode. Only the last two components have been carried out. For testing the machine, loading and linking is performed by an ad-hoc program. This is illustrated in Figure 4.

The specification written in B is about 10,000 lines long. To this part developed formally according to the B method, it is needed to add components

[7] Developed in the framework of the european project IST MATISSE, number IST-1999-11435.

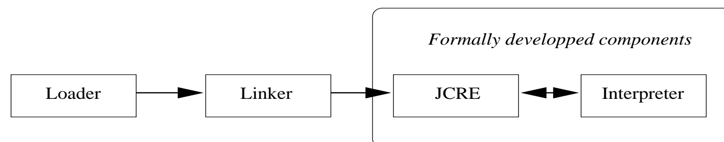

Fig. 4. Components of the Java Card Virtual Machine

developed outside the method, either to implement basic machines, or to interface the B components to low-level devices, or yet to simulate the underlying operating system (memory management, etc.). These components written in C are approximatively 5,000 lines long. One may consider that it constitutes a medium-size formal case study[8], but it is significant in the world of smart cards and embedded applications.

5.2 Analysis of the case study

In project BOM experiments, we compared four translation chains from B to C:

1. the translator provided by the actual version 3.6 of Atelier B,
2. a BOM translator without optimization and with a general set of translation rules. It translates data and programs in a standard manner and is referred to as BOM(G) (for *General*) in the paper,
3. a BOM translator, adapted for smart cards by the way of some specific translation rules, called BOM(C) (for *Cards*) and possibly optimizing the generated code by inlining, as explained in Section 3.2.
4. a prototype developed by Gemplus in the framework of the MATISSE project, called Simple C, dealing with a limited subset of B_0. Like BOM(C), it is adapted for smart cards but it does not perform inlining.

The aim of the fourth translator was to assess the feasibility of code generation for smart cards. The Java Card bytecode verifier has been embedded on a smart card [9, 7] using this latter translator. The comparison of the results of the translations will be done in Section 5.4 with respect to these four translators. The components of the case study are presented in Figure 5. One can distinguish the following parts:

1. "Loader and linker" are the first modules (written in C) of the JCVM.
2. "Interface between B and C" is constituted of some C code which ensures the links between the C modules and the programs generated from the B developments.
3. "B components" is the part developed with the B method. It contains the JCRE module and the interpreter of Java Card bytecode.
4. "Basic B machines" are the machines which are no further developed in B. They provide interfaces with integer types and with system primitives.

[8] By comparing to industrial B developments which can reach 100,000 lines of B code.

5. “Basic C implementations” are the implementations of the basic B machines.
6. “Hardware abstraction” is the model in C of hardware components needed for the case study.
7. “Atelier B runtime environment” is the set of C code used by Atelier B to execute the components translated in C by the actual translator. This module is needed only for the Atelier B translator.

The “loader and linker” and the “Hardware abstraction” parts do not depend on the various translator versions of the case study. They will not be considered in the remainder of this paper. The “Interface between B and C” part differs between Atelier B, in which multi-instancing of machines is allowed, and the other translators. The case study contains nine basic B machines. They are implemented directly by C programs. Notice that the basic machines for integers do not generate specific C code. The “B components” part contains thirty components (twelve machines, twelve implementations and six refinements). Only implementations differ, due to the adaptation of the B_0 types.

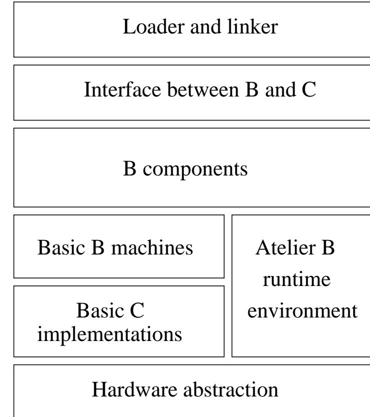

Fig. 5. Components involved in the case study

5.3 Compilation chain and characteristics of the target platforms

The translation chain can be represented as in Figure 6. “B files” and “Written by hand C files” are provided by the developer. The other files are generated by the tools. The size comparisons are done at the stage “Compiled files”, i.e. size in byte of the object code. At this stage, the size of the various components is accessible for a detailed analysis (which is not the case in the binary file).

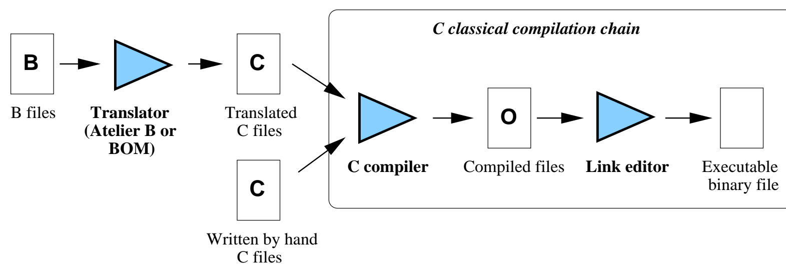

Fig. 6. Translation schema

Besides the four translation chains, we have experimented the translation for two target platforms with very different characteristics. By this way we can assess the efficiency of the BOM translator in several contexts. The chosen platforms are:

- An Atmel AVR 8-bit RISC microcontroler[9]. For this platform, the compiler used is the IAR one[10], and the optimizations are set to minimize code size. The code size presented corresponds to the size of the code given by the compiler plus the size of the constant data.
- The SmartMIPS[11], a 32-bit MIPS processor specially designed for smart cards. It is more representative of high-end next generation smart cards. For this platform, the compiler used is the MIPS SDE version of the gcc compiler. The mips16 instruction set is used, and the optimizations are set to reduce code size. For this platform, the size provided corresponds to the size of the "text" segment as given by the `sde-size` utility.

5.4 Comparisons between the translators

The tables in Fig. 7 and Fig. 8 summarize the results of the code sizes generated by the various translators with respect to the two platforms. A detailed comparison is available in a BOM project report [21]. The measures are given respectively for the Atmel AVR and the SmartMIPS platforms. For each column c, the "Gain" line displays the value: $(total(AtelierB) - total(c)) * 100/total(AtelierB)$.

Atmel AVR	Atelier B	BOM(G)	BOM(C)	SimpleC
B Components	12,692	4,870	4,326	4,492
Basic B machines	2,970	140	140	677
Basic C implementations	726	726	726	726
Interface	1,606	24	24	20
Atelier B runtime env.	1,140			
Total	17,994	5,760	5,216	5,915
Gain	0 %	68 %	71 %	67 %

Fig. 7. Code sizes (bytes) for each translators (Atmel AVR platform)

In the case of the AVR platform, the BOM translators clearly outperforms the Atelier B one. The difference of code size between "Interface" and "Basic B machines" of table 7 is mainly due to the fact that the Atelier B translator allows multi-instancing of machines. Indeed, this feature requires to introduce in the generated code an internal representation (using C structures) of machine states. Therefore, machine initialization needs dynamic allocation and associated initialization code, and machine states are accessed using indirections. The

[9] URL: http://www.atmel.com/products/AVR/.
[10] URL: http://www.iar.com.
[11] URL: http://www.mips.com/products/s2p12.html.

difference between translated B machines is also partly due to the same reason. But the benefits of BOM translators and Simple C on the B components also come from the use of extended B_0 types (as explained in section 4.2), which are especially well suited to 8-bit platforms such as the Atmel AVR one.

SmartMIPS	Atelier B	BOM(G)	BOM(C)	SimpleC
B Components	9,686	7,630	5,018	6,916
Basic B machines	2,286	168	168	720
Basic C implementations	1,034	1,040	1,040	1,032
Interface	624	42	42	36
Atelier B runtime env.	588			
Total	14,218	8,880	6,268	8,704
Gain	0 %	38 %	56 %	39 %

Fig. 8. Code sizes (bytes) for each translator (SmartMIPS platform)

As the SmartMIPS is a 32-bit platform, the Atelier B code is more adapted, and differences between the translators are tighter. Contrary to the AVR platform, using small integer types is less efficient than using the default 32-bit integer. Thus, both the Atelier B translator as well as the BOM translator uses 32-bit integers, reducing the gap between the translators. However, the overhead implied by the initialization of the machines still remains, and can be seen on the "Interface" and "Basic B machines" lines of table 8.

Another comparison can be done, even if they rely upon a little part of the case study. The component JCRE was written in C for the Gemplus prototype translator, because this translator was not able to deal with some B features used to specify this component. So, it is possible to compare the size of this component (after compilation) with the size of the translations from B to C of the same component. The summary is given in Fig. 9. The line "Overhead", in each column c, displays the value: $(size(c) - size(1)) * 100/size(1)$.

Atmel AVR	C implementation	BOM(C)	BOM(G)	Atelier B
Size JCRE	537	596	634	1,704
Overhead	0 %	11 %	18 %	217 %
SmartMIPS				
Size JCRE	536	588	652	904
Overhead	0 %	10 %	22 %	69 %

Fig. 9. Overhead with respect to a C program

5.5 Main results

Let us sum up now the results of this case study. BOM translators produce more efficient code than Atelier B. It is however not surprising, because Atelier B is not very optimized. BOM translators produce code which efficiency is comparable with the output of SimpleC, for which some parts are directly encoded in C due to the incompleteness of the prototype. It is an interesting result because Gemplus experiments have shown that SimpleC is compliant with smart cards requirements [10, 9]. Moreover, BOM translator has several important advantage over SimpleC. Indeed, SimpleC is an incomplete prototype which is not integrated with Atelier B. Moreover it does not offer any mechanism to adapt the translation process.

In regard to the points which are detailed in this paper, the B_0 type extension is interesting for code efficiency, as illustrated in the case of the Atmel AVR platform. Moreover, adding finer types at the level of the B_0 language offers a set of guarantees due to the proof process, in particular when operations without overflow are used. The main improvement in code efficiency is achieved by the optimization phase as illustrated in Fig. 10 for the part "B components".

	BOM(G)	BOM(C) without inlining	Gain	BOM(C) with inlining	Gain
AVR size	4,870	4,820	1.03 %	4,326	11.17 %
SmartMIPS size	7,630	7,572	0.76 %	5,018	33.60 %

Fig. 10. Code sizes (bytes) with and without optimizations

Abstract models generally introduce operations to modularize specifications and the proof process, as explained in Section 3.2. Thus, inlining can improve implementations in a significant way, depending on the choice of the calls to be inlined. In the case study the gain is important because the interpretor abstract model contains a lot of operations which are called only once. After inlining, about a hundred of operations have been eliminated. Recall that inlining is valid only when condition 1 (Section 2.5) is fulfilled. In the case study, no restriction has been met by this condition.

Finally the adaptability of BOM translation rules allow a developer to optimize the translator for a given platform and for a given project. Indeed, some translation rules corresponding to a frequent form of code in the project can be made more efficient. In the case study, apart for C types, only few adaptations have been necessary.

6 Conclusion

The objectives of the BOM project were to carry out a translation chain from B specifications to C programs. The imposed constraints were that the generated

code had to be embedded in smart cards as it would be done if the application was directly written in C. Moreover, the chain should guarantee that the code is correct with respect to the high level specifications and it is runtime error free. At last, the translator should be adaptable to various target platforms and compilers.

In this paper, we focused the attention on the techniques introduced in the translation chain to achieve adaptability and embedding. Others results of the BOM project are the definition of an operational semantics of the B_0 language in order to establish the total correctness of the translation process and a generalization of the adaptability of the B_0 language for other languages or types. These results are under development. With respect to the purpose of software embedding, the (light) restrictions imposed to the B language were sufficient to implement a translator which generates a reasonably compact code. The optimization phase, although relatively simple, decreases significantly the code size. Comparisons of Section 5.4 show a promising result, since the overhead compared to manually translated C code is reduced to around 10% using the BOM translator. This overhead, although it cannot be neglected, still remains acceptable, especially if the benefits of using formally proved code are taken into account. With respect to adaptability, two techniques were considered useful, implemented and tested on the case study. The adaptation of the translation rules needs more complete experiments, specially for their impact on the execution time for some specific platforms. On the contrary, adaptation of the integer types to basic C types has been found very convenient for example, for the 8-bit AVR platform.

So, the project results are satisfactory and demonstrate the ability to generate automatically executable code which is comparable to a code written by hand with ordinary programming languages. The gains rely then on the guarantees provided by the use of a formal method and on the certification level which can be obtained by this way. As far as we know, only few formal methods support code generation which is as time/space efficient as handwritten code. In [17] an alternative approach is developed: rather than proving in advance that the translator always produces a target code which correctly implements the source code (translator verification), each individual translation (i.e. run of the translator) is verified. A key feature of this validation is its full automation. Nevertheless, such an approach seems not possible in the B framework, due to the generality of the considered programs.

References

1. J.-R. Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, August 1996.
2. P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A Successful Application of B in a Large Project. In *FM'99 - Formal Methods, LNCS 1708*, pages 369–388. Springer-Verlag, 1999.
3. P. Behm, L. Burdy, and J-M Meynadier. Well Defined B. In D. Bert, editor, *B'98 : The 2nd Int. B Conference, Proceedings, LNCS 1393*. Springer-Verlag, 1998.

4. D. Bert. Étude de la traduction B0 vers C : Conventions de nommage. Technical Report http://lifc.univ-fcomte.fr/~tatibouet/WEBBOM, Rapport D10, projet RNTL BOM, 2003.
5. D. Bert, M.-L. Potet, and Y. Rouzaud. A Study on Components and Assembly Primitives in B. In H. Habrias, editor, *Proceedings of the 1st Conference on the B method*, pages 47–62. IRIN Institut de Recherche en Informatique de Nantes, 1996.
6. L. Burdy. Traitement des expressions dépourvues de sens de la théorie des ensemble. Application à la méthode B. Thèse de Doctorat, CNAM, 2000.
7. L. Burdy, L. Casset, and A. Requet. Développement formel dun vérifieur embarqué de byte-code Java. *Technique et Science Informatiques (TSI): Développement rigoureux de logiciel avec la méthode B*, 22, 2003.
8. L. Casset. *Construction Correcte de Logiciels pour Carte à Puce.* PhD thesis, Aix-Marseille II University, Marseille, France, October 2002.
9. L. Casset. Development of an Embedded Verifier for Java Card Byte Code using Formal Methods. In L.-H. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe (FME)*, volume LNCS 2391, pages 290–309, Copenhagen, Denmark, 2002. Springer-Verlag.
10. L. Casset, L. Burdy, and A. Requet. Formal Development of an embedded verifier for Java Card Byte Code. In *International Conference on Dependable Systems & Networks (DSN)*, pages 51–58, Washington, D.C., USA, June 2002. IEEE Computer Society.
11. L. Casset and J.-L. Lanet. How to formally specify the Java Bytecode semantics using the B method. pages 1–8, Lisbon, Portugal, June 1999.
12. ClearSy. B Language Reference Manual, version 1.8.5. Technical report, ClearSy System Engineering, URL :http://www.clearsy.com/, 2001.
13. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In Jean-Jacques Quisquater and Bruce Schneier, editors, *CARDIS*, volume LNCS 1820, pages 85–97. Springer-Verlag, 2000.
14. C. Morgan. *On the Refinement Calculus.* Springer-Verlag, 1992.
15. S. Motré. A B automaton for Authentification Process. In *WITS: Workshop on Issues in the Theory of Security*, Genève, Suisse, 2000.
16. S. Motré and C. Téri. Using Formal and Semi-Formal Methods for a Common Criteria Evaluation. In *EUROSMART*, Marseille, France, 2000.
17. A. Pnueli, M. Siegel, and O. Shtrichman. Translation Validation for Synchronous Languages. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. of the 25th International Colloquium on Automata, Languages and Programming (ICALP 1998), LNCS 1443*, pages 235–246. Springer-Verlag, 1998.
18. M-L. Potet. *Spécifications et développements formels: Etude des aspects compositionnels dans la méthode B.* Habilitation à Diriger des Recherches, INPG, 2002.
19. M.-L. Potet and Y. Rouzaud. Composition and Refinement in the B method. In D. Bert, editor, *B'98 : The 2nd Int. B Conference, LNCS 1393*. Springer-Verlag, 1998.
20. A. Requet. A B Model for Ensuring Soundness of the Java Card Virtual Machine (Extended Version). *Science of Computer Programming, Elsevier Science*, 46(3):283–306, 2003.
21. A. Requet. Évaluation du traducteur C. Technical Report http://lifc.univ-fcomte.fr/~tatibouet/WEBBOM, Rapport D11, projet RNTL BOM, 2003.
22. International Standard. Programming languages — C. ISO/IEC 9899:1999 (E), 1999.