CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 240 (2009) 79–96
www.elsevier.com/locate/entcs

Verified Compilation and the B Method: A Proposal and a First Appraisal

Bartira Dantas

Universidade do Estado do Rio Grande do Norte
Natal, RN, Brazil

David Déharbe[1,2] Stephenson Galvão
Anamaria Martins Moreira Valério Medeiros Júnior

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Natal, RN, Brazil

Abstract

This paper investigates the application of the B method beyond the classical algorithmic level provided by the B0 sub-language, and presents refinements of B models at a level of precision equivalent to assembly language. We claim and justify that this extension provides a more reliable software development process as it bypasses two of the less trustable steps in the application of the B method: code synthesis and compilation. The results presented in the paper have a value as a *proof of concept* and may be used as a basis to establish an agenda for the development of an approach to build *verifying compilers* [11] based on the B method.

Keywords: Verifying compiler, B method.

1 Introduction

The construction of certified software requires the application of special techniques such as correct-by-construction approaches. Some examples are the extraction of programs from mathematical proofs of the satisfiablity of their specification [9] and obtaining code, either from successive proved correct refinements [4] or from *ad hoc* refinements defined by the designer and formally verified *a posteriori*. The B method [2], used as a basis for the work described in this paper, follows the latter approach, although the results obtained are extensible to the former approach.

[1] Partly financed by CNPq projects n 485576/2007-4 and 307597/2006-7.
[2] Email: david@dimap.ufrn.br

1571-0661 © 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.05.046

The B method manages to successfully span the development process from modelling down to the algorithmic level. However, to cover the remaining steps towards a running implementation, one needs to synthesize the algorithmic model to some programming language and then compile the resulting code into the target platform assembly language. These last two steps cannot be verified using the formal verification approach provided by the B method. Indeed, code synthesis maps constructs of languages that do not have common semantic underpinnings. Compilation is even more troublesome, since, in addition to the semantic gap, there is also usually a deep transformation of the code structure caused by optimisation and other transformations, and the effort put in the B-based development may be jeopardized by a bug in the compiler. To increase the confidence on the generated code, industrial adopters of the B method employ a redundant tool chain, using two distinct implementations of code synthesis and compilation. This pragmatic approach however does not provide a theoretically satisfying evidence of the result correctness and comes at the cost of having to deal with the execution of redundant programs.

This paper proposes an approach that, although based on existing ingredients, innovates in employing them to extend the reach of the B method to the assembly level, thereby eliminating the need for these two less reliable steps following the application of the B method, namely code synthesis and compilation. Proof obligations may be generated and discharged to check that the resulting assembly program refines, i.e. is a correct implementation of, the original functional model. Employing this approach, the translation to binary code is as simple and direct as that of an assembler, i.e. does not require modification of the code structure and may be considered as a (trivial) implementation of a one-to-one function from assembly symbolic instructions to their binary correspondents. The compilation presented in this paper has been performed manually, and the experience gained will be used to define translation rules and implement them in a tool.

Related work.

The approach investigated in this paper requires to verify that the result of the compilation is a correct refinement for each possible input. Another approach is to prove once for all the correctness of the compiler itself. Such proof relies on the formalization of the input and output languages in a unique semantic model. This approach has been undertaken successfully in several research projects (e.g. [16] [7] [13] [14] [6] [3]). In these approaches, the input languages of the verified compiler are programming languages, or subset thereof, and the target languages are assembly languages. The semantics of both languages, as well as possible intermediate representations, and the different translation steps are formalized in the logics of proof assistants [15,5,19]. In order to build a certified compiler for B using such approach, one would first need to develop a full formalization of the B method in a mechanical proof assistant, formalize the B0 constructs, the target platform(s) and the compilation of the high level constructs in terms of the target platform instruction set.

[3] See [10] for an extensive bibliography on compiler verification.

Plan of the paper.

The paper is structured as follows. Section 2 provides a general introduction to the B method and briefly outlines the proposed approach, comparing it to a classical application of the B method. Section 3 presents the main lines of the definition of a B model for the instruction set of a simple, yet computationally representative, micro-controller. Section 4 presents the mapping of the main constructs of the algorithmic language B0 to assembly constructs, through a series of simple examples, and introduces thus the structure of an assembly program as a B implementation. Experimental results of the application of this approach to commercial micro-controllers are also reported in this section. Finally, Section 5 draws preliminary conclusions on this work and an agenda for future research in the direction of constructing a production-level verifying compiler based on the B method.

2 Introduction to the B method

The B method for software development [2] is based on the B *Abstract Machine Notation* (AMN) and the use of formally proved refinements up to a model sufficiently concrete that programming code can be automatically generated from it. Its mathematical basis is first order logic, integer arithmetic and set theory, and its constructs are similar to those of the Z notation [18]. Its structuring constructs are more closely related to imperative modular programming language constructs. Also, its more restrictive constructs simplify the job of support tools, and industrial software for the development of B based projects is widely available [8,3].

2.1 Overview

A B specification is modular: a module defines a set of valid states, including initial states, and operations that may lead to a state transition. The design process starts with a module with a so-called *functional* model of the system under development. In B, such module is called a *machine*. The B method requires to prove the satisfiability of the model constraints, that the initial states are valid and that operations do not map valid states to invalid states.

Once an initial functional model has been constructed and verified, the B method provides constructs to define *refinement* modules. A refinement is always associated to another, more abstract, model and specifies a design decision: either about the concrete representation of the state, or about the algorithmic realization of an operation (or both). The B method imposes that the user proves that each refinement conforms to the refined module.

Finally, so-called *implementation* modules form a special case of refinement where the abstraction level is similar to that of a programming language. This paper uses the term *algorithmic model* to qualify such modules. The part of the B notation that may be used to define implementations is called B0 (e.g. it does not contain non-deterministic constructs). Using as input an implementation module, it is possible to generate source code in a conventional program language such as C or ADA.

The B method is illustrated graphically in Figure 1. At each step, proofs need to be developed to check the consistency of the model or the conformity of the refinements. Once the resulting code has been compiled, it may be verified using test data generated from the initial functional model. This verification is generally not exhaustive but may detect errors introduced by the code synthesis tool or the compiler.

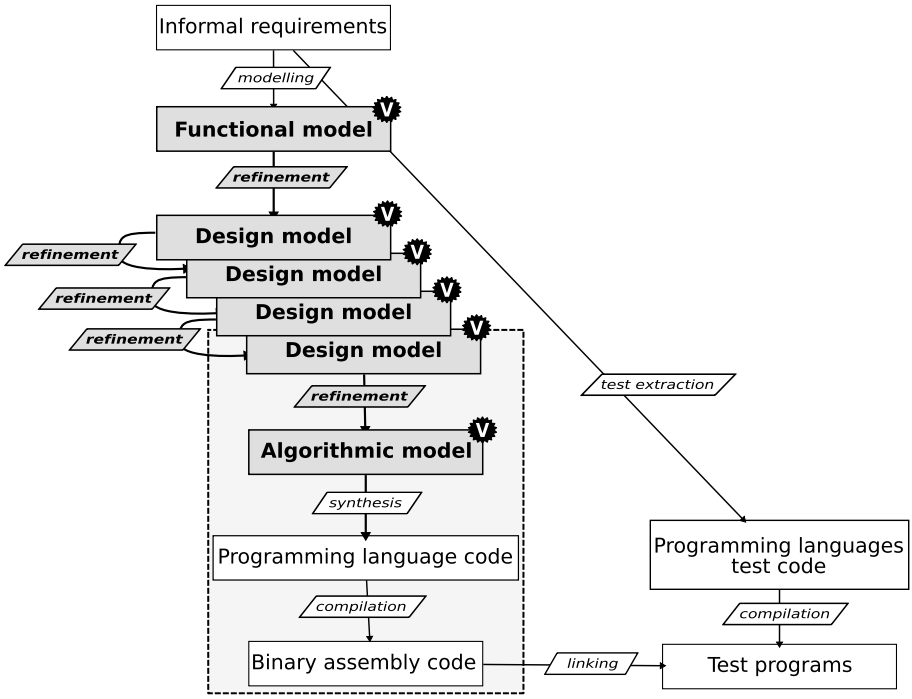

Figure 1. Overview of a software engineering process based on the B method. Rectangle correspond to different artifacts. Slanted rectangles are human or automated activities. Grey rectangles are artifacts produced in the B method. The V labels emphasize that formal verification is applied to the corresponding artifact. The light-grey area is the main focus of this work.

2.2 The B notation

Essentially, a B module contains two main parts: a state space definition and the available operations. It may additionally contain auxiliary clauses in many forms (parameters, constants, assertions), but those are essentially for practical purposes (i.e. to promote modularity, reuse, etc.) and do not extend the expressive power of the notation. In the remainder, we will restrict our discussion to the core clauses of the module specification.

The specification of the state components appears in the VARIABLES and INVARIANT clauses. The former enumerates the state components, and the latter defines restrictions on the possible values they can take. If V denotes the state variables of a machine, the invariant is a predicate on V. Verifications carried out throughout the development process have the intention of checking that no invalid

state will ever be reached as long as the operations of the machine are used as specified.

For the specification of the initialisation as well as the operations, B offers a set of so-called *substitutions*. These are "imperative-like" constructions with translation rules that define their semantics as the effect they have on the values of any (global or local) variables to which they are applied. The semantics of the substitutions is defined by the *substitution calculus*, formalizing how the different substitution forms rewrite to formulas in first-order logic. Let S denote a substitution, E an expression, then $[S]E$ denotes the result of applying S to E. For instance, an operation that would increment a counter variable v can be specified as $v := v + 1$. Indeed, the basic substitution is very similar to the side-effect free assignment construct found in imperative programming languages. Applying such substitution to an expression consists in substituting the target variable v with the source expression. For instance, $[v := v+1]v \geq 0$ simplifies to $v+1 \geq 0$. Besides the basic substitution, the B method provides more elaborate substitution constructions, such as:

- Non deterministic substitution **ANY** v **WHERE** C **THEN** S **END** applies substitution S with variable v having any value that satisfies predicate C.
  Substitution $v :\in V$, where V is a set, is equivalent to **ANY** x **WHERE** $x \in V$ **THEN** $v := x$ **END**.
- Parallel substitution $[S \parallel S']$ applies both substitutions S and S' simultaneously.
- The substitution with pre-condition **PRE** C **THEN** S **END** is used to specify a partial operation, defined only when condition C holds. For instance, the operation that increments v only when it is smaller than value *top* may be specified as **PRE** $v < top$ **THEN** $v := v + 1$ **END**.

The example shown in Figure 2 illustrates the most basic clauses in the functional model of a traffic light. The model is named *traffic_light*, as defined in clause MACHINE. The state is composed of a single variable, named *color*. The value of this variable must belong to *COLOR* and is non-deterministically initialized with one element of this set. Follows then the specification of the operation *advance*, modelling a transition of the traffic light.

The previous model is then refined to a module where the state is a single integer value (Figure 3). The VARIABLES clause declares the unique component state in the refinement, and its relationship with the functional model state is established in the INVARIANT clause: the value of *count* is equal to the application of function *color_refine* to the abstract variable *color*.

2.3 Overview of the approach

The *weakest link* in a software production process based on the B method is the synthesis of software in a programming language and its compilation towards the target platform assembly language. As the B0 language is close to programming constructs, code synthesis is usually considered safe; however if the target language uses constructs unsupported by the B0 language (e.g. object orientation), this transformation may not be as straightforward as it seems. In industrial practice, a

MACHINE *traffic_light*
SETS $COLOR = \{green, yellow, red\}$
VARIABLES *color*
INVARIANT $color \in COLOR$
INITIALISATION $color :\in COLOR$
OPERATIONS
advance =
  CASE *color* **OF**
    EITHER *green* **THEN** $color := yellow$
      OR *yellow* **THEN** $color := red$
      OR *red* **THEN** $color := green$
    END
  END
END

Figure 2. An example of a functional model in B

REFINEMENT *traffic_light_data_refinement*
REFINES *traffic_light*
CONSTANTS *color_refine*, *color_step*
PROPERTIES
  $color\_refine \in COLOR \rightarrow \mathbb{N} \wedge$
  $color\_refine = \{green \mapsto 0, yellow \mapsto 1, red \mapsto 2\} \wedge$
  $color\_step \in 0..2 \rightarrow 0..2 \wedge color\_step = \{0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0\}$
VARIABLES *count*
INVARIANT $count \in \mathbb{N} \wedge count \in 0..2 \wedge count = color\_refine(color)$
INITIALISATION $count := 0$
OPERATIONS
  $advance = count := color\_step(count)$
END

Figure 3. A B refinement module for the example of Figure 2

redundant tool chain (two code synthesis tools and two compilers) is used to produce two versions of the program. Both versions are executed and their results should agree. This approach requires twice the number of computational resources that would be necessary if only one instance of the generated program was executed. To reduce this overhead, one could consider adopting a code synthesis targetting the source language of a verified compiler. However one would still need to close the semantic gap between this language and B0.

This paper proposes a new approach to address these issues, by applying the concepts of the B method to generate software artifacts down to assembly level. The approach is divided into: (1) modelling the target computational platform, and (2) refining the algorithmic model to an implementation solely based on the platform model.

The target platform may be modeled with the B abstract machine notation: the state of the machine represents the state of the platform (i.e. registers and memory), and each operation represents an assembly instruction. This only needs to be performed once for a given target platform. Further details are provided in Section 3, where a model of the Random Access Machine model of computation is

presented.

The algorithmic model has to be further refined into an assembly-level model. The latter model is defined on top of the target platform model discussed previously. A general strategy for this refinement is to map the state variables of the algorithmic model to different addresses of the platform memory, and to translate the algorithmic-level operations to combinations of operations defined in the platform model corresponding to the assembly language instructions. The resulting assembly-level refinement needs to be proved compliant with the corresponding functional model. We then obtain a software artifact at the assembly level that formally refines the initial functional model.

This approach provides an extension to the B method as sketched in the left of Figure 4. However, the classic B method has some restrictions that prevent us from applying this ideal "strategy", namely that loop constructs may only appear in algorithmic models, and such modles may not be suject to refinements.

Fortunately, it is possible to devise another solution to circumvent this limitation without modifying or extending the B method. This solution is illustrated at the right of Figure 4: instead of establishing a refinement relation between the assembly and the algorithmic models, consider a refinement of the design model directly preceding the algorithmic model in the refinement process. The construction of the assembly implementation from the algorithmic implementation should be formalized by a set of rules that can be implemented to form a B-based formal compiler, but the verification would be carried out as usual in a B refinement, with respect to the design model. Section 4 provides a series of small examples, representative of the different kinds of algoritmic constructs provided by the B method, and shows the correspondence between algorithmic and assembly implementations. The examples have been produced manually, with the aim to gain insight on how the translation could be formalized into a set of rules that would serve as the basis of a compiler from B0 to assembly-level models. They all use the assembly-like platform presented in the following.

3 A B model for an assembly language

In this section, we present the functional model of a minimal processor. Its actual definition in the B notation is interspersed with an informal description of the model.

The essence of current micro-processors and micro-controllers is captured by the Random Access Machine computation model [12], which is Turing complete. Therefore, we use and model this machine. The variables composing the state of the machine are *mem*, an unbounded memory storing bounded natural numbers, *pc*, the program counter, and *end*, that is used to detect the end of the computation. When executing a program on the *ram* processor, the value of the program counter is less or equal than that of the end marker; when it is equal, then the execution stops. The machine can be initialized in any state.

Each instruction of the *ram* processor is modelled as a B operation. Instruction *nop*, the so-called no-operation (also known in B as *skip*) increments the program

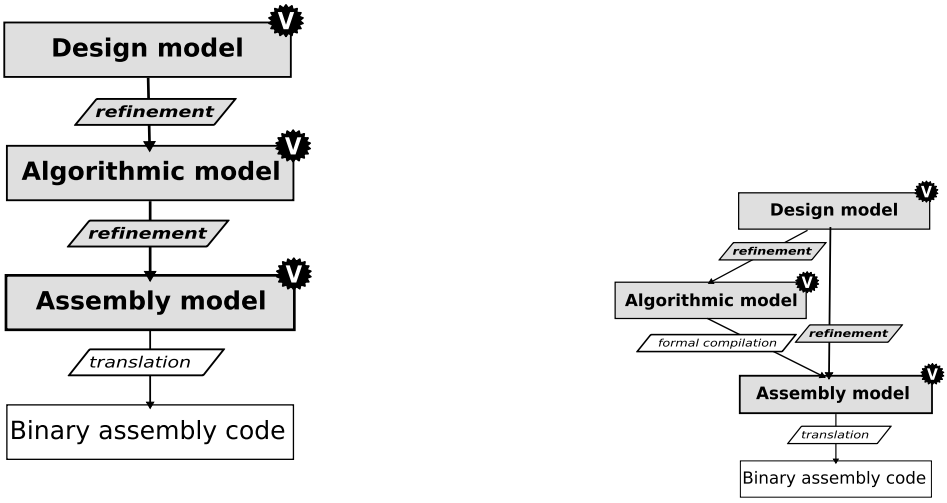

Figure 4. Extending the B method down to the assembly level: ideal (left) and actual (right).

counter as its only effect. Instruction *set* is responsible for storing a constant value *val* into a memory location *a*. Observe that the execution of an instruction has a side effect on the program counter. Instruction *inc* is the only data modifying instruction in our machine: it increments the value stored at the location *a* of the memory. Observe that the set $\mathbb{N}$ is bounded in B, and the increment instruction is restricted so that overflow does not occur. Instruction *copy* copies the value stored in address *src* to the address *dest*. Instruction *testgt* is a conditional branch: it tests if the value stored in location *a1* is strictly greater than the value stored in location *a2*. If this is the case, the program counter is incremented by one, otherwise two. Instruction *testeq* (omitted here) is very similar but tests for equality. The *goto* instruction directly alters the value of the program counter. It is an unconditional branch, and is used in combination with the testing instructions to implement conditional and iteration constructions. Moreover, before starting the execution of a program, one needs to reset the program counter and the end of program marker. This functionality is provided by the operation *init*, that takes as input the size of the program, i.e. the number of instructions. This is the necessary condition for the *ram* processor to be able to start executing instructions.

This approach has been applied by the authors to build formal B models of the programming resources (or part thereof), offered by several representative commercial micro-controllers, namely Microchip's PIC, Intel's 8051 and Zilog's Z80.

MACHINE ram
CONCRETE_VARIABLES mem, pc, end
INVARIANT $mem \in (\mathbb{N} \rightarrow \mathbb{N}) \wedge pc \in \mathbb{N} \wedge end \in \mathbb{N}$
INITIALISATION $mem :\in (\mathbb{N} \rightarrow \mathbb{N}) \parallel pc :\in \mathbb{N} \parallel end :\in \mathbb{N}$
OPERATIONS
$init(sz) =$ **PRE** $sz \in \mathbb{N}$ **THEN**
$pc := 0 \parallel end := sz$
END;
$nop =$ **PRE** $pc + 1 \leq end$ **THEN** $pc := pc + 1$ **END**;
$set(a, val) =$ **PRE** $a \in \mathbb{N} \wedge val \in \mathbb{N} \wedge pc + 1 \leq end$ **THEN**
$mem(a) := val \parallel pc := pc + 1$
END;
$inc(a) =$ **PRE** $a \in \mathbb{N} \wedge mem(a) < max(\mathbb{N}) \wedge pc + 1 \leq end$ **THEN**
$mem(a) := mem(a) + 1 \parallel pc := pc + 1$
END;
$copy(src, dst) =$ **PRE** $src \in \mathbb{N} \wedge dst \in \mathbb{N} \wedge pc + 1 \leq end$ **THEN**
$mem(dst) := mem(src) \parallel pc := pc + 1$
END;
$goto(val) =$ **PRE** $val \in \mathbb{N} \wedge val \leq end$ **THEN** $pc := val$ **END**

$testgt(a1, a2) =$ **PRE** $a1 \in \mathbb{N} \wedge a2 \in \mathbb{N} \wedge$
$(mem(a1) > mem(a2) \Rightarrow (pc + 1 \leq end)) \wedge$
$(mem(a1) \leq mem(a2) \Rightarrow (pc + 2 \leq end))$
THEN
IF $mem(a1) > mem(a2)$ **THEN** $pc := pc + 1$ **ELSE** $pc := pc + 2$ **END**
END;
END

4 Compilation of B0 constructs: examples

The modelling language of the B method has a sub-language, named B0, for algorithmic models. Variable assignment as well as sequential, conditional and iterative composition form the basic constructs of B0. This section shows, through a series of increasingly complex (yet basic) examples, how these constructs can be refined as combinations of assembly instructions of the *ram* processor presented in Section 3.

4.1 Sequencing

The design model in the machine *sequencing* has two variables and a single operation that swaps the values of these variables.

VARIABLES a, b
INVARIANT $a \in \mathbb{N} \wedge b \in \mathbb{N}$
INITIALISATION $a :\in \mathbb{N} \parallel b :\in \mathbb{N}$
OPERATIONS
$run =$ $a := b \parallel b := a$

The salient parts of a possible algorithmic model are given below. The variables are implemented in a machine *nat* that provides setter and getter for a variable holding a value of the set *NATURAL* (a bounded set of natural numbers).

IMPORTS $ai.nat, bi.nat$
INVARIANT $a = ai.value \wedge b = bi.value$
OPERATIONS
$run =$
VAR tmp **IN**
$tmp := bi.value$;
$bi.set(ai.value)$;
$ai.set(tmp)$
END

Compilation to assembly includes mapping the variables to memory locations and algorithmic-level instructions to sequences of assembly instructions. Here, variables *ai* and *bi* are mapped to locations 0 and 1, respectively. This static memory allocation is formalized in the invariant of the assembly-level model. In the presented examples, the types of the model variable and of the memory locations match, so *the assembly invariant is obtained by substitution of the implementation variables by their corresponding memory location in the algorithmic invariant*. For this example, we have:

IMPORTS $uc.ram$
INVARIANT $a = uc.mem(0) \wedge b = uc.mem(1)$

The compiled assembly program is a sequence of three *copy* instructions that perform the swap between the memory locations corresponding to the model variables. The assembly-level operation is thus:

OPERATIONS
$run =$
BEGIN
$uc.init(3)$;
$uc.copy(1, 2)$;
$uc.copy(0, 1)$;
$uc.copy(2, 0)$
END

Although this model is a valid refinement of the initial machine, we will show, in the next example, that its construction pattern cannot be used as a general solution to build assembly-level refinements.

4.2 Conditional statement

This new example system has two operations: one to register values and one to return the largest registered value. The state is the set of all the values registered so far.

VARIABLES st
INVARIANT $st \in \mathbb{P}(\mathbb{N})$
INITIALISATION $st := \emptyset$
OPERATIONS
$add(v) =$
PRE $v \in \mathbb{N} \wedge v \notin st$ **THEN**
$st := st \cup \{v\}$
END;
$res \longleftarrow largest =$ **PRE** $st \neq \emptyset$ **THEN** $res := \max(st)$ **END**

This is a classical example where an abstract variable may be refined to a variable of a simpler data type. We focus our attention on the refinement of the first operation:

```
IMPORTS mi.nat
INVARIANT max(st) = mi.value
OPERATIONS
  add(v) =
  IF mi.value < v THEN
    mi.set(v)
  END
```

The assembly model may no longer be a sequence of assembly instructions as in the previous example. The reason is that the execution flow is no longer linear, due to the conditional construct. To cope with this situation, the assembly model is built as the execution model of the *ram* machine: based on the current value of the program counter, an instruction is fetched and executed. This action is repeated until the program counter reaches the end of the program:

```
IMPORTS uc.ram
INVARIANT max(st) = uc.mem(0)

OPERATIONS
  add(v) =
  BEGIN
    uc.init(3);
    WHILE uc.pc < uc.end DO
    BEGIN
      CASE uc.pc OF
        EITHER 0 THEN uc.set(1, v)
               OR 1 THEN uc.testgt(1, 0)
               OR 2 THEN uc.copy(1, 0)
        END
      END
    END
    INVARIANT ...(see below)
    VARIANT ...(see below)
    END
  END
```

This strategy needs a loop construct that halts when the program counter reaches the end marker. We call this the *fetch loop*: it associates each possible value of the program counter with the corresponding assembly instruction. The local variable *pc* maintains the value of the program counter of the *ram* machine. The local variable *end* stores the end marker of the program and remains constant. Both variables are employed in the formulation of the variant and invariant of the loop. As the algorithmic model does not jump backwards, the variant can be expressed as the distance between the end of the program and the value of the program counter. The invariant of the fetch loop establishes the relationship between the variables of the *ram* machine and the local variables of the operation and the state of the *ram* memory for each possible valuation of the program counter:

INVARIANT
$0 \leq uc.pc \wedge uc.pc \leq uc.end \wedge$
$(uc.pc = 0 \Rightarrow uc.mem(0) = max(st)) \wedge$
$(uc.pc = 1 \Rightarrow uc.mem(0) = max(st) \wedge uc.mem(1) = v) \wedge$
$(uc.pc = 2 \Rightarrow uc.mem(0) = max(st) \wedge uc.mem(1) = v \wedge max(st) < v) \wedge$
$(uc.pc = 3 \Rightarrow uc.mem(0) = max(v, max(st)))$
VARIANT $uc.end - uc.pc$

4.3 Loop

The last example implements the addition of two integers as iterated increments.

VARIABLES a, b, s
INVARIANT $a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge s \in \mathbb{N} \wedge (a + b) \in \mathbb{N}$
OPERATIONS
$run = s := a + b$

Recall that the *ram* machine only has an increment operation, and addition needs to be implemented iteratively, as in the following algorithmic model:

IMPORTS $ai.nat, bi.nat, si.nat$
INVARIANT $a = ai.value \wedge b = bi.value \wedge s = si.value$
OPERATIONS
$run =$
VAR $partial, i$ **IN**
$partial := ai.value;$
$i := 0;$
WHILE $i \neq bi.value$ **DO**
BEGIN
$partial := partial + 1;$
$i := i + 1$
END
INVARIANT $i \in \mathbb{N} \wedge i \leq bi.value \wedge partial = ai.value + i$
VARIANT$(bi.value - i)$
END;
$si.set(partial)$
END

Based on this algorithm, we devise the following assembly-level model of the addition algorithm by iterated increments (excerpts):

INVARIANT $a = uc.mem(0) \wedge b = uc.mem(1) \wedge s = uc.mem(2)$
OPERATIONS
$run =$
BEGIN
$uc.init(7);$
$i := 0;$
WHILE $uc.pc < uc.end$ **DO**
BEGIN

```
    CASE uc.pc OF
      EITHER 0 THEN uc.copy(0, 2)
           OR 1 THEN uc.set(3, 0)
           OR 2 THEN uc.testeq(1, 3)
           OR 3 THEN uc.goto(7)
           OR 4 THEN uc.inc(2)
           OR 5 THEN uc.inc(3)
           OR 6 THEN BEGIN
                       uc.goto(2)
                       i := i + 1
                     END
      END
    END
  END
  INVARIANT
    0 ≤ uc.pc ∧ uc.pc ≤ uc.end ∧ i ∈ ℕ ∧
    uc.mem : (ℕ → ℕ) ∧ uc.mem(0) = a ∧ uc.mem(1) = b∧
    (uc.pc = 0 ⇒ i = 0 ∧ uc.mem(2) = s)∧
    (uc.pc = 1 ⇒ i = 0 ∧ uc.mem(2) = a)∧
    (uc.pc = 2 ⇒ (uc.mem(2) = a + uc.mem(3) ∧ uc.mem(3) = i ∧ i ≤ b))∧
    (uc.pc = 3 ⇒ (uc.mem(2) = a + uc.mem(3) ∧ uc.mem(3) = i ∧ i = b))∧
    (uc.pc = 4 ⇒ (uc.mem(2) = a + uc.mem(3) ∧ uc.mem(3) = i ∧ i < b))∧
    (uc.pc = 5 ⇒ (uc.mem(2) = a + uc.mem(3) + 1 ∧ uc.mem(3) = i ∧ i < b))∧
    (uc.pc = 6 ⇒ (uc.mem(2) = a + uc.mem(3) ∧ uc.mem(3) = i + 1 ∧ i < b))∧
    (uc.pc = 7 ⇒ (uc.mem(2) = a + uc.mem(3) ∧ uc.mem(3) = i ∧ i = b))
  VARIANT pgvar(uc.pc, uc.mem(1), i)
  END
END
```

This assembly model follows the same pattern as the previous example. Instructions 0 and 1 codify the preamble, 2 to 6 the increment loop, and 7 is the end of the program execution. Note that we need a variable, here called i, to keep track of the number of times the increment loop has been executed. Its value is initially zero and it is incremented whenever the program counter is 6, i.e. when the algorithm jumps back to the evaluation of the loop condition. Also, observe that the invariant of the fetch loop states that when the value of the program counter is two, the algorithmic loop condition shall hold.

The variant is the value of a function application to the three program elements *pc*, the program counter, *uc.mem(1)*, the value of b, and i, the number of times the increment iteration of the algorithm has been executed. The function is called *pgvar* (program variant) and is defined in the following machine:

CONSTANTS $pgvar, pgsz, la, lsz$
PROPERTIES
$pgsz \in \mathbb{N} \wedge la \in \mathbb{N} \wedge lsz \in \mathbb{N} \wedge$
$pgvar \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N} \wedge$
$pgsz = 7 \wedge lsz = 5 \wedge la = 2 \wedge$
$\forall pc, b, i \bullet$
$(pc \in \mathbb{N} \wedge b \in \mathbb{N} \wedge i \in \mathbb{N} \Rightarrow$
$(pc = 7 \Rightarrow pgvar(pc, b, i) = pgsz - pc) \wedge$
$((pc = 0 \vee pc = 1) \Rightarrow$
$pgvar(pc, b, i) = pgsz - pc + lsz \times b) \wedge$
$((pc = 2 \vee pc = 3 \vee pc = 4 \vee pc = 5 \vee pc = 6) \Rightarrow$
$pgvar(pc, b, i) = pgsz - la + lsz \times (b - i) - (pc - la)))$

The expression of the function body is quite complex and, for the sake of clarity, uses three numeric constants: *pgsz* the total number of instructions in the program, *lsz* the size of the block codifying the loop and *la* the address of its first instruction. The value returned by *pgvar* is greater or equal to the number of instructions that remain to be executed. The corresponding expression needs to be derived from the loop variant of the algorithmic level model.

4.4 Lessons learnt

The three examples presented form a basic, yet fully expressive, language. These assembly models have been built manually and the refinement relation between these models and the initial design models has been checked using the B4free tool through the Click'n'Prove interface [1]. Note that, in general, it is not possible, due to the rules of the B method, to establish a B refinement from an algorithmic model to an assembly model. Our intuition is that it would be easier, as the semantic gap is narrower.

In the general case, deriving the assembly program from the algorithmic models involves applying classical compilation techniques to map algorithmic variables and instructions to memory locations and a sequence of assembly instructions respectively. We have seen that the elements of a verifiable assembly level refinement are:

- The model invariant formalizes the mapping of variables to memory locations, and is obtained by applying, in the algorithmic invariant, substitution of the algorithmic variables by the corresponding memory locations,
- Each model operation is a fetch loop that associates the possible values of the program counter with the corresponding assembly instructions,
- Fetch loop invariants specify the state of the assembly machine for each possible value of the program counter. It may be derived from the model invariant, the semantics of assembly instructions, and the possible loop invariants of the algorithmic model.
- Fetch loop variants express an upper bound of the number of instructions that remain to be executed, for each possible value of the program counter. The expression of such functions may be derived using static code analysis such as worst-case execution time techniques [17].

Model	Program size	Different instructions	Proof obligations	
			automatic	interactive
PIC	12	6	155	5
8051	9	5	167	3
z80	17	7	564	5

Table 1
Statistics on implementations of the traffic light on different micro-controllers

Based on the insight obtained from this first appraisal, a second experiment was made, namely to apply the approach investigated to actual computing platforms. This experiment is briefly commented in the next section.

4.5 Experiment with commercial micro-controllers

Based on the conclusions of this preliminary study, the relevant part of the instruction set of three commercial micro-controllers was modelled following the style of the RAM machine (Section 3). The selection of micro-controllers over micro-processors was motivated by the ultimate goal of employing the approach to produce safety-critical embedded software.

The chosen products were Microchip's PIC16C432, Intel's 8051, and Zilog's Z80, three micro-controllers widely used in the industry that are therefore good candidates as target in a full-fledged investigation of the approach presented in this paper. It is interesting to note that the B projects of the different micro-controllers share some basic definitions useful in hardware modelling (such as the definition of bit vector type and operations, and their conversion to integer ranges).

The traffic light example of Section 2.2 was then manually compiled to assembly-level B models in each of these platforms. Table 1 reports some statistics about these projects: the size of the assembly programs, the number of different assembly instructions used in these programs, and the number of generated proof obligations that were verified automatically and interactively. Initially, a large number of proof obligations would not be proved automatically by the theorem prover provided with the tool support. After examination of these proof obligations, we observed that they required instantiating the definition of the functions defining the increment of the program counter. We then added a few lemmas stating simple facts about this function (B provides an "assertions" clause to introduce such lemmas) and most proof obligations were then discharged automatically by the prover.

4.6 Open issues.

The study presented in the paper does not cover some important aspects such as: data structures (including pointers), modularity, libraries, input/output, interrupts, real-time, etc.

All the examples only manipulate simple integer or scalar values. A full-scale project would certainly involve more complex data structures that can be described in the B0 language. On the other hand, it is worth noting that, in order to avoid certain classes of bugs, software in safety-critical projects is often restricted so that it does not use features such as dynamic memory allocation and pointers.

Support for modularity should result in techniques that cater calls to routines in such a way that when a formally verified assembly-level refinement is available, this result can be leveraged to show the correctness of the caller routine. This criterion is fundamental to ensure the scalability of the approach.

Micro-controllers provide a number of facilities (i.e. dedicated hardware), such as input/output and interrupt pins for interfacing directly the processor with its environment, timers, etc. In the presented examples, such facilities have not been taken into account. Future work include modelling and using such facilities in the B development.

5 Conclusion and future work

In this paper, we identified a potential way to apply the B method to solve the challenge of "the verifying compiler". We have presented the results of an initial appraisal of the feasability of this approach.

Applying the B method, we built a formal model inspired from the Random Access Machine, a computational model that is similar to that of mainstream micro-controllers and micro-processors. Using as a target the model of this machine, we developed three assembly level B implementations of increasing complexity that employ four basic, yet computationally complete, algorithmic constructs: assignment, sequence, conditional and loop. We obtained thus a template for assembly level models for the B method. This template shall be instantiated by a mapping between algorithm variables and the machine memory addresses, the compilation of the algorithm in a sequence of assembly instructions, the construction of the assembly program variant and invariant, based on the memory layout mapping, the algorithmic model invariants and variants, and the assembly instruction semantics.

This approach has been instantiated to build correct assembly implementations of a simple operation onto different commercial micro-controllers. Although the B method may not be adequate to formalize all aspects of a microcontroller specification, we found it quite suitable to provide a formal model of the assembly language instructions.

This initial appraisal shows the feasibility of the B-based approach to build a verifying compiler, thereby taking advantage of a large body of techniques and tools to address this grand challenge.

However, much work remains to be done to meet the goals of building a full-fledge tool chain based on the B method and targetting an assembly-level industrial platform. We shall define and implement formal rules for the construction of assembly level models from algorithmic models. The application of such rules would result in a set of proof obligations that would need then to be verified. In order

to address expected scalability issues in this verification effort, different solutions may be investigated. First, the restrictions imposed by B method on refinements could be relaxed, so that the translation may be realized in small steps, using intermediate models mimicking intermediate representation of compilers, where issues such as register allocation and code optimization are more easily implemented and verified (in a way similar to that of, e.g. [14]). Also, some proof obligations need manual assistance with the current tool support for B. Another line of work is to investigate if and how the automation of these provers can be improved (by developing and applying additional proof rules). Another solution would be to generate proof scripts corresponding to these proof obligations along the translation process.

More work also needs to be done from the modelling viewpoint. On the one hand, more elements of the B0 language need to be addressed, allowing for richer data structures. In order to cater to designers of safety-critical systems, one also needs to provide support for facilities such as interrupts, timers and input/output. In this line, modelling and integrating real-time operating systems services is also worthy of attention.

Eventually, the experience gathered in this research should be used to provide tool support for developpers. This support shall be integrated with existing frameworks for the B method, or variations thereof, and should automate the generation of assembly models and the verification of the corresponding refinement relations.

References

[1] Abrial, J.-R. and D. Cansell, *Click'n prove: Interactive proofs within set theory*, in: *TPHOLs*, 2003, pp. 1–24.

[2] Abrial, R., "The B-Book: Assigning programs to meanings," Cambridge University Press, 1996.

[3] B-Core Ltd, *The b-toolkit*, http://www.b-core.com/btoolkit.html.

[4] Back, R.-J. and J. Wright, "Refinement Calculus: a Systematic Introduction," Springer, 1998.

[5] Bertot, Y. and P. Castran, "Interactive Theorem Proving and Program Development (Coq'Art: The Calculus of Inductive Constructions)," Texts in Theoretical Computer Science, Springer, 2004.

[6] Blazy, S., Z. Dargaye and X. Leroy, *Formal verification of a c compiler front-end*, in: *Proc. Formal Methods* (2006), pp. 460–475.

[7] Bowen, J. P., J. He and P. Pandya, *An approach to verifiable compiling specification and prototyping*, in: P. Deransart and J. Maluszynski, editors, *Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science **456** (1990), pp. 45–59.
URL citeseer.ist.psu.edu/bowen90approach.html

[8] Clearsy, *Atelier B*, http://www.atelierb.eu.

[9] Constable, R., "Implementing Mathematics with the Nuprl Development System," Prentice Hall, 1986.

[10] Dave, M. A., *Compiler verification: a bibliography*, SIGSOFT Softw. Eng. Notes **28** (2003), pp. 2–2.

[11] Hoare, C. A. R., *The verifying compiler, a grand challenge for computing research*, in: *VMCAI*, 2005, pp. 78–78.

[12] Hopcroft, J. E. and J. D. Ullman, "Introduction to Automata Theory, Languages and Computation," Addison Wesley, 1979.

[13] Leinenbach, D., W. Paul and E. Petrova, *Towards the formal verification of a c0 compiler: Code generation and implementation correctnes*, in: *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods* (2005), pp. 2–12.

[14] Leroy, X., *Formal certification of a compiler back-end or: programming a compiler with a proof assistant*, SIGPLAN Not. **41** (2006), pp. 42–54.

[15] Matt Kaufmann, P. M. and J. S. Moore, "Computer-Aided Reasoning: An Approach," Kluwer Academic Publishers, 2000.

[16] Moore, J. S., *A mechanically verified language implementation*, J. Autom. Reason. **5** (1989), pp. 461–492.

[17] Park, C., *Predicting program execution times by analyzing static and dynamic program paths*, Real-Time Systems **5** (1993), pp. 31–61.

[18] Spivey, J., "The Z Notation: a Reference Manual," Prentice-Hall International Series in Computer Science, Prentice Hall, 1992, 2nd edition.

[19] Tobias Nipkow, M. W., Lawrence C. Paulson, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic," LNCS, Springer, 2002.