Development of Formal Components using the B method

Dorian Petit[1,2], Vincent Poirriez[2], and Georges Mariano[1] *

[1] INRETS ESTAS, 20, rue Élisée Reclus
F-59650 VILLENEUVE D'ASCQ - France,
[dorian.petit,georges.mariano]@inrets.fr,
[2] Université de Valenciennes et du Hainaut Cambrésis, LAMIH/ROI,
F-59313 Valenciennes Cedex 9, France,
[dorian.petit,vincent.poirriez]@univ-valenciennes.fr

Abstract. The aim of this paper is to merge two approaches of software development. The first one is the component approach. Developping software components is now a new challenge in the software industry. The second approach is the formal one. These approaches are not so distant if we consider Bertrand Meyer's opinion: a component without contracts can not be reused (more exactly, he said that it was more complicated to reuse such a component). One of the difficulties with the design by contract approach is to find the contracts. In some formal approach -we will use the *B* method in this paper- the software properties (the contracts) are expressed in the specifications. We present in this paper a tool we have developped to generate code in the spirit of the component approach from *B* specifications. Thus, we will see how we can link the notion of component and the *B* specifications.

1 Introduction

The use of formal methods in software development increase the quality and the reliability of software. In some cases -when a dysfunction of the system can lead to human or financial losses- formal methods should be used (see the common criteria [1]). We will not tell more on this subject, because it was already discussed in the literature ([2] and [3] for example).

The component-based development is an important approach in software development. Some of the aims of this approach are near to those of formal development. The common main advantage is that the quality of the software increases: if you use a component-based approach to develop a software, you focus on a piece of the software, and consequently you make less mistakes.

Bowen and Hinchey give in [4] some recommendations to use formal methods in the industry. These recommendations are expressed in ten commandments. Two of them are of particular interest for our work: the second and the ninth.

* This work have been partially supported by the COLORS french project (COmposants LOgiciels Réutilisable et Sûr which means Reusable and Reliable Component Software), a TACT 2000 project

The second says : “Thou shalt formalize but not over-formalize”. This commandment expresses that it is “both unnecessary and costly” to apply formal methods to all parts of a system. The authors speak about the “components of the system that will benefit from a formal treatment”. The notion of component seems to be natural when you use formal methods.

The ninth commandment says: “Thou shalt test, test, and test again”. Even if a formal method is used to develop all the system, it is necessary to test.

Considering more “traditional” (as opposed to formal) development methods, a new approach seems to be interesting: “design by contract”. This approach is based on adding assertions in the software. In his book *Object-Oriented Software Construction* ([5]), Bertrand Meyer mentions that an assertion expresses what the software should do and the code how it does.

Assertions are a valuable contribution to the quality of the software because:

- they help in the production of correct and robust software
- they help to document the code
- they help to test the code
- they serve as the basis for exception handling

The reuse is one of the main characteristic of component-based development. The main conclusion of [6] -article on the lessons of the Ariane 5 crash- is that “reuse without a precise specification mechanism is a disastrous risk”. Using the design by contract approach is a way to solve this problem.

To summarize, the three approaches -component-based development, formal methods and design by contract- are complementary. The B method is not currently adapted to mix the three approaches. The aim of our article is an experiment to fill the gap between the B method and the two other development methods.

To illustrate what we propose, we use a toy example: the bounded stack. A bounded stack is simply a stack with a limited number of items.

The article is organised as follows. In section 2, we present briefly the B method and the algorithm we use to transform the B specifications into a format allowing to generate the code (section 3). Finally, in section 4, we discuss about the advantage of the mixing on the B development process in section 4.

2 The B method

B ([7]) is a formal development method used in industry to product software for safety-critical systems. This method covers the entire development process: from abstract specifications to implementations.

2.1 Refinement in B

One of the mechanism used in B is the refinement. It is used to transform step by step an abstract specification into a concrete representation. This concrete

representation, called implementation in B terminology, can be translated into a programming languages (currently C, C++ and Ada).

To give a simple example of what a refinement is, figure 1 presents a specification of the `push` operation of the bounded stack.

```
push(addval) =
  PRE stack_top < stack_size ∧ addval ∈ NAT
  THEN
    stack_top := stack_top + 1 ||
    the_stack(stack_top + 1) := addval
  END;
```

Fig. 1. The push operation: abstract specification

This operation can be refined by the one presented in figure 2.

```
push(addval) =
  BEGIN
    stack_top := stack_top + 1 ;
    the_stack(stack_top) := addval
  END;
```

Fig. 2. The push operation: concrete specification

The refined operation retrieves the indeterministic due to the `||` constructor.

2.2 B Component

B specifications can be decomposed in several sub problems (as we do in classic software development). Different mechanisms can be used to link the sub-problems. They are called "link clauses". Each clause (`INCLUDES`, `IMPORTS`, `SEES` and `USES`) has its specific semantic: they are used to share different parts of the specifications.

The term component is not used in B as it is usually. In the B context, a component is a formal text, a "machine" (the most abstract form of specifications), a "refinement" (intermediate form of specifications) or an "implementation" (the most concrete form of specifications). While usually the term component denotes a software entity. The B term equivalent is module. A B module is the assembly of one machine, zero one or more refinements and one implementation[3].

[3] the other kinds of module will not be take into account here because they are not used during the code generation phase

To be unambiguous, we will use the term component in its usual meaning and B component to denote the B specification.

2.3 Flattening B specifications

Flattening B specifications consists in eliminating the refines and the composition links. From a set of B components, the flattening algorithm build only one B component equivalent to the initial set of B components. All the information of the different specifications are merged into one formal text.

On our example, the operation `push` in the flatten B component will be the following: the flatten precondition is the result of the concatenation of the

```
push(addval) =
  PRE
    stack_top < stack_size ∧ addval ∈ NAT ∧
    the_stack ∈ (1..stack_size) −− > NAT ∧ stack_top ∈ NAT ∧
    stack_top ≥ 0 ∧ stack_top ¡= stack_size
  THEN
    BEGIN
      stack_top := stack_top + 1;
      the_stack(stack_top) := addval
    END /*BLOCK*/
  END /*PRE*/
```

Fig. 3. The push operation: the flatten version

abstract precondition and the abstract invariant. The body of the operation is the concrete one.

This notion of flattening exist implicitly in the BBook ([7]). The term flattening also appears in M.L. Potet and Y. Rouzaud works ([8]). But it is in the S. Behnia's PhD ([9], in French) that the algorithm is entirely specified. Our tool uses this specification of the algorithm.

2.4 Proof obligations

For each refinement step and for each composition link are associated some proof obligations to guarantee the correctness of the development.

The work necessary to discharge all the proof obligations is the part who takes the most time in a B development. The most the B project is decompose, using refinement or composition, the easiest the proof are. So, the flattening algorithm has to be used, a priori, subsequently to the proof stage. But, we will see in section 4.2 how we can use it before this proof stage.

3 Code Generation

3.1 Generation Process

Our code generation can be divided in two steps. The first one uses the flattening algorithm to gather all the information into only one B component. The second step transforms this B component into the desired code using an XLST processor. The code generation process is illustrated in Fig. 4.

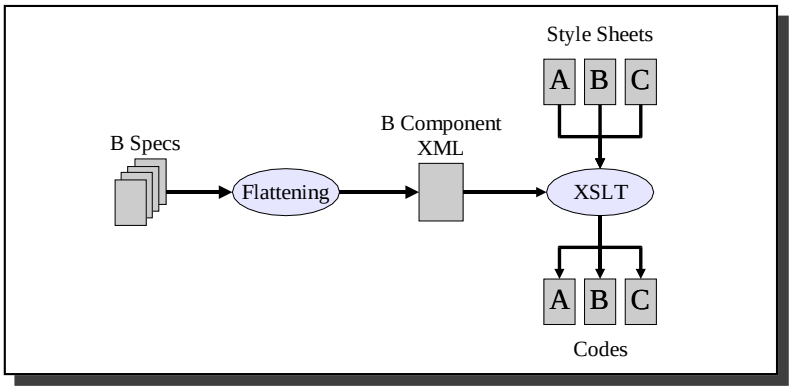

Fig. 4. Code generation process

From several B specifications, the flatten B component is built. Using our tool, this flatten B component is in an XML ([10]) form. This is useful to simplify the rewriting step, we just have to use an XSLT ([11]) processor and the appropriate style-sheet. To change the target language, we just have to change the style-sheet.

3.2 Component generation

To generate a component from B specifications, you just need to select the set of specifications you want to flatten. The flattening algorithm can help you to choose because some verifications -on the dependencies between the specifications- are made before flattening.

An advantage of using the flattening algorithm is that the properties of the specifications are preserved and can be used to add assertions in the component we generate. In Eiffel ([12]) -the language who has one of the most developed assertions mechanisms-, there exists four kinds of assertions:

- `require`, the preconditions in the methods
- `ensure`, the postconditions in the methods
- `invariant`, for classes or loops invariants

– `check`, to verify properties in a method

The precondition and invariant constructions exist in the B language, we can directly transform and insert them in the code. In a recent extension of the B method ([13]), the postcondition construction is added. There is no construction in B that have the same semantic as the `check` one.

Figure 5 presents the generated code for the push operation.

```
method push addval =
    require (stack_top ≥ 0 ∧ stack_top ≤ stack_size) ;
    require (stack_top < stack_size ∧ stack_top ≥ 0 ∧
             stack_top ≤ stack_size) ;
    (
      stack_top := stack_top + 1;
      BASIC_Sets.mul_set the_stack stack_top addval
    )
      ;
    ensure (stack_top ≥ 0 ∧ stack_top ≤ stack_size) ;
```

Fig. 5. The push operation: the code

4 Advantages

4.1 Simplicity

Using XML/XSLT technologies makes easier the code generation. The intermediate representation of the component (the flatten B component) is syntactically unambiguous, so the transformation is easier to write. You just have to specify the translation between the B abstract syntax and the target language syntax. You can also compose different style-sheets. This can be useful if, you have to use your component on two different machines (a smart-card and a smart-card terminal for example). You can decompose the treatment in two style-sheets: one for the control statements of the target language and another one more specific for the data type (who can be different) for example.

If you want to validate the process, it is easier if the process is divided into several small parts. And you can replace the use of an XSLT processor by another rewriting processor in which you have more confidence.

4.2 Impact on the B development process

To explain the impact our approach can have, we should remind some basis of the B development process. The B development consists in translating the informal specifications into abstract ones. Then, these specifications are refined and

decomposed. The proof stage can lead to some modifications in the specifications (if a proof obligation can not be discharged).

If we annote the code, we can find some errors earlier. This is the purpose of some works on static or dynamic checking: [14], [15] or [16].

Applying such an approach to a B development is equivalent to find a counter-example that shows that there is a problem in the specifications. This is in the same way as J. Rushby in [17]: "Disappearing Formal Methods". The software industry has developed and used mechanisms to validate/test their software. It will be unpopular and prejudical to totally forget their knowledges, especially if our solution is (or seens to be) more difficult.

Figure 6 illustrates this approach. On the right side, you find the classical B approach, extended with ours on on the left. On the left side, you can used whatever you want to validate the generated code (and consequently the specifications). You can do this before, in parallel or after the proof of the project.

You can then continue the normal development and use classical tools. But you can also use our code generation tools to take the benefit of component-based tools validation.

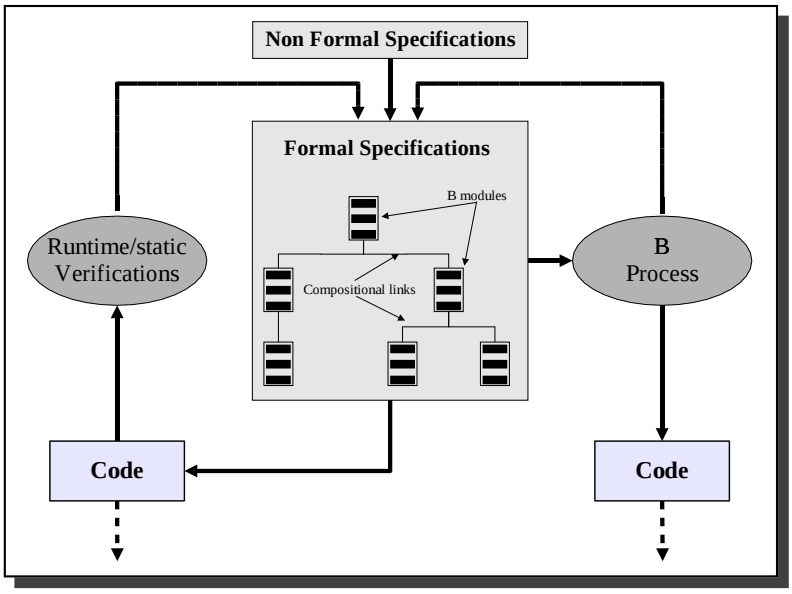

Fig. 6. Development process

4.3 Impact on the integration of the generated code

This impact is the direct consequence of using the contract and component approaches. It is easier to integrate a contract-component than a generated code

using classical B tools, because these approaches are especially developed for that.

To generate annoted code allows to verify the properties you have verified during the proof stage. This can seem useless, but what about executing the code you have totally proved to be correct on a bugged processor ? In that case, checking the properties at runtime can be helpful.

5 Experimentation

Our tool has been experimented on a specification of a bounded stack (cf appendix A for the specifications). Appendix B presents the resulting code.

For the code generation, we have chosen the OCaml language as the target language ([18]) because this language is implicitly typed. That allows us to ignore the type information from the specifications. However, that will not induce technical problems, but we search to show the feasibility of the proposed process.

To take advantage of the type inference of the OCaml language, we are induced to retrieve the type information from the assertions (these information are in the B specification, either in the invariant, or in the preconditions).

From the generated code, we note several points to comment:

- The sets management implies the use of the `BASIC_Sets` module. It is necessary to supply some basic components reusable by the generated code (as in the classical tools for the code generation).
- The management of the return values of the operations can not be managed in OCaml as in B. We use a OCaml writing trick to do this, that's why we have abstracted it in the `Resultat` module.
- In the code, the `require` and `ensure` constructions correspond to the definitions of the assertions. As the invariant notion does not exist in OCaml, we append it to the `require` and the `ensure` assertions. In our example, there is a redundancy with the invariant. This is due to the flattening mechanism of the `REFINES` link. This is not the case with the other links. For the loop invariants, we can put a `require` construction in the loop.

6 Conclusions

This article presents an approach and a tool to generate annoted software components from B specifications. The tool is based on the flattening algorithm and the use of an XSLT processor to translate a B component into a component in the desire language. We did not speak about the interfaces of the component we generate. It is easy to imagine that the same process we use to generate the component code can be used to generate the appropriate interfaces with its contract (as in the approach of [19]). Our approach allows to extend easily the set of the possible target languages. In our experimentation, we have chosen to add the OCaml language in the potential target languages.

We can also generate code annoted with the properties we extract from the B specifications. This allows B users to mix a formal approach (proofs) and a more classical development method (tests and runtime checking). This is clearly in the same way that Meyer et al in [20]. More trust can be put in an annoted component than in a component generate with the B tools, because there is a (potential, because you can turn off the assertions verifications) redundancy of verifications.

The code generation is not any more dependent on the $B0$ (the more concrete B language). The B language we use in the component we generate is the gathering of all the B languages (abstract, concrete). So this notion of concrete language disappears and we can stop the refinement process earlier than when we use the other B dedicated tools. This can be useful if your target language -OCaml for example- have sequences constructions, you do not have to refine the sequences into tabular (or something at the same low level).

References

1. Common Criteria: Common criteria for information technology security evaluation (1999) Part 3 : Security Assurance Requirements, version 2.1.
2. Hinchey, M.G., Bowen, J.P., eds.: Applications of Formal Methods. Series in Computer Science. Prentice Hall International (1995)
3. Hall, A.: Seven myths of formal methods. IEEE Software **7** (1990)
4. Bowen, J.P., Hinchey, M.G.: Ten commandments of formal methods. IEEE Computer **28** (1995) 56–63
5. Meyer, B.: Object Oriented Construction. Prentice-Hall (1988)
6. Jézéquel, J.M., Meyer, B.: Design by contract: The lessons of Ariane. Computer IEEE **30** (1997) 129–130
7. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (1996)
8. Potet, M.L., Rouzaud, Y.: Composition and refinement in the B method. In Bert, D., ed.: B'98 : The 2nd International B Conference. Volume 1393 of Lecture Notes in Computer Science (Springer-Verlag)., Montpellier, LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer Verlag (1998) 46–65
9. Behnia, S.: Test de modèles formels en B : cadre théorique et critères de couvertures. Thèse de doctorat, Institut National Polytechnique de Toulouse (2000)
10. World Wide Web Consortium (W3C): (Extensible markup language (XML)) http://www.w3.org/XML/.
11. World Wide Web Consortium (W3C): (XSL transformations (XSLT)) http://www.w3.org/TR/xslt/.
12. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
13. MATISSE: Event B reference manual. (2001)
14. Leino, K.R.M.: Extended static checking: A ten-year perspective. In: Informatics. (2001) 157–175
15. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML. Technical Report 02-05, Department of Computer Science, Iowa State University (2002) To appear in SERP 2002.

16. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software (2002)
17. Rushby, J.: Disappearing formal methods. In: High-Assurance Systems Engineering Symposium, Albuquerque, NM, Association for Computing Machinery (2000) 95–96
18. Leroy, X.: The Objective Caml system Documentation and user's manual version 3.04. Institut National de Recherche en Informatique et Automatique. (2001)
19. Schmidt, H.: Trusted components: Towards automated assembly with predictable properties. (In: ICSE CBSE4)
20. Meyer, B., Mingins, C., Schmidt, H.: Providing trusted components to the industry. IEEE Computer **5/1998** (1998) pp104–105

A Bounded stack B Specifications

```
MACHINE
        stack(stack_size)

DEFINITIONS
        empty == stack_top = 0

CONSTRAINTS
        stack_size : NAT & stack_size >= 1 & stack_size <= MAXINT

VISIBLE_VARIABLES
        the_stack, stack_top

INVARIANT
        the_stack : (1..stack_size)-->NAT &
        stack_top : NAT & stack_top >= 0 & stack_top <= stack_size

INITIALISATION
        the_stack :: (1..stack_size) --> NAT || stack_top := 0

OPERATIONS
        flag <-- is_empty = flag := bool(empty);

        push(addval) =
        PRE stack_top < stack_size & addval : NAT
        THEN
          stack_top := stack_top + 1 ||
          the_stack(stack_top + 1) := addval
        END;

        pop =
        PRE not (empty)
        THEN
          stack_top := stack_top -1
        END;
```

```
        result <-- top =
        PRE not (empty)
        THEN
          result := the_stack(stack_top)
        END
END
```

```
IMPLEMENTATION
        stack_1(stack_size)


DEFINITIONS
        empty == stack_top = 0

REFINES
        stack

INITIALISATION
        the_stack := (1..stack_size)*{0};
        stack_top := 0

OPERATIONS
        flag <-- is_empty = flag := bool(empty);

        push(addval) =
        BEGIN
          stack_top := stack_top + 1 ;
          the_stack(stack_top) := addval
        END;

        pop =
        BEGIN
          stack_top := stack_top -1
        END;

        result <-- top =
        BEGIN
          result := the_stack(stack_top )
        END
END
```

B Bounded stack generated code

```
open Modules
class stack stack_size =
  object
    val mutable the_stack = (BASIC_Sets.mul
                                (((BASIC_Sets.make_range 1 stack_size)),
                                 (BASIC_Sets.make_compr [0] )))
```

```
  val mutable stack_top = 0

  method is_empty =
    let flag = Resultat.init () in
    require ( stack_top ≥ 0 ∧ stack_top ≤ stack_size ) ;
    flag := Resultat.set bool(stack_top = 0) ;
    ensure (stack_top ≥ 0 ∧ stack_top ≤ stack_size ) ;
    ( (Resultat.out flag))

  method push addval =
    require ( stack_top ≥ 0 ∧ stack_top ≤ stack_size ) ;
    require (stack_top < stack_size ∧ stack_top ≥ 0 ∧
             stack_top ≤ stack_size ) ;
    (
      stack_top := stack_top + 1;
      BASIC_Sets.mul_set the_stack stack_top addval
     )
      ;
    ensure (stack_top ≥ 0 ∧ stack_top ≤ stack_size ) ;

  method pop =
    require ( stack_top ≥ 0 ∧ stack_top ≤ stack_size ) ;
    require ( ¬ (stack_top = 0) ∧ stack_top ≥ 0 ∧
              stack_top ≤ stack_size ) ;
    (
      stack_top := stack_top − 1
     )
      ;
    ensure (stack_top ≥ 0 ∧ stack_top ≤ stack_size ) ;

  method top =
    let result = Resultat.init () in
    require ( stack_top ≥ 0 ∧ stack_top ≤ stack_size ) ;
    require ( ¬ (stack_top = 0) ∧ stack_top ≥ 0 ∧
              stack_top ≤ stack_size ) ;
    (
      result := Resultat.set the_stack.(stack_top)
     )
      ;
    ensure (stack_top ≥ 0 ∧ stack_top ≤ stack_size ) ;
    ( (Resultat.out result))
end
```