APPLICATION EXPLORATION OF B METHOD IN THE DEVELOPMENT OF SAFETY-CRITICAL CONTROL SYSTEMS

LIU NING[1], WANG KEMING[2], HOU XILI[3], WANG XIA[2] & CHENG PENG[2]
[1]Graduate School of Tangshan, Southwest Jiaotong University, China
[2]School of Information Science and Technology, Southwest Jiaotong University, China
[3]TongHao GBA (GuangZhou) Smart Control Co., Ltd., China

ABSTRACT

This article presents our experience in developing a tram control system using the B formal method. We find that there are some notable issues when using an abstract machine model to express software systems and in automatic code generation, for which we have summarized the solutions. In this paper, we illustrate how to use the B module to develop complex systems, the rational choice of implication relations of the invariants, as well as the correctness of the variable definition in the model for code generation. The solutions to these issues can help developers with less experience to better use the B method to develop the reliable systems.

Keywords: B method, invariant, modelling, code generation, application exploration.

1 INTRODUCTION

Using formal methods to develop the safety-critical system can greatly reduce the design defects caused by the system requirements specification and the misinterpretation by developers, which consequently improves the safety of the system [1].

The B method has been successfully applied to the industry of transportation [2], and there are some successful cases [3]. Our group used the B method on the development of the tram control system of Guangzhou Huangpu Line 1 in China. To the best of our knowledge, this is the first time for the B method to be applied to develop the tram control system in China.

It remains a major challenge for both researchers and engineers to develop a complex industrial system with formal methods. Scientific theoretical knowledge and engineering empirical knowledge are equally important for applying formal methods. The former helps developers apply formal methods to develop software systems, and the latter helps developers better apply formal methods, but few articles share the experience.

This article focuses on the experience of using B method modelling and code generation. Due to their lack of background knowledge and engineering experiences of formal methods, some developers encounter many notable problems with their applying B method of engineering practice. A period of time for the past we spent on developing the Guangzhou tram project and encountered many difficulties in the using of the B method. We also met some typical issues such as developing complex software systems with the B module, the choice of implication relations of the invariants as well as the definition of variables in the model for code generation, etc. We believe that how to deal with these issues mentioned above is critical to the successful application of the B method on engineering projects.

The remainder of this paper is organized as follows: The B method is briefly introduced in Section 2. Section 3 demonstrates how to use the B module to develop complex systems and the rational choice of implication relations of the invariants by cases. Section 4 discusses two inappropriate forms of modelling. In Sections 5 and 6, we discuss some related work and give the conclusion of this paper.

WIT Transactions on The Built Environment, Vol 199, © 2020 WIT Press
www.witpress.com, ISSN 1743-3509 (on-line)
doi:10.2495/CR200261

2 B METHOD

As a software development method, the basic idea of the B method is to provide a representation method that uses the Abstract Machine Notation (AMN) and the Generalized Substitution Language (GSL) to describe software. The software described by the B method can eventually be implemented in imperative programming languages, or even in assembly language [4]. The method provides a framework in which specifications can be refined through to implementations, which can be translated into a programming language. The B method is the first single formal method that covers all stages of the software development process from specification to design to implementation [5].

The basic unit for describing software with the B method is an abstract machine, which includes: data description (constants, variables); operation description (a set of operations on data); invariants (a set of relationships that the data must satisfy) [6], these abstract machines describe the most basic requirements. Developers understand software systems in the manner of an abstract machine model, that is, a state and various operations that can modify this state. This analysis method constitutes a study of their static and dynamic. The static corresponds to the definition of states, while dynamic correspond to various operations [4].

Besides, the B method supports refinement, that is, the process of modifying from an abstract specification to a concrete implementation through some data or operation substitution. Implementation is a special case of refinement and can be done at any stage in refinement, but may be done only once. The implementation cannot have any private state, and have to implement these operations using the specified operations of other machines imported into the implementation [5]. When using the executable code of the B method to implement the system, the system must be refined into implementation [7].

The supporting platform tool for the B method is Atelier B [8]. After the developer builds and saves the B model, the model must pass the type-checker before it can generate proof obligations and prove. The ProB [9] also provides an interface for Atelier B to link.

3 PROPERTIES VERIFICATION BASED ON INVARIANTS

B method states that after describing the operation, developers need to prove the invariant to ensure that this operation has not broken the consistency of the abstract machine. This requires developers have to understand how to use the abstract machine model to express software systems, as well as be able to choose the implication relations of the invariants correctly. In the B method, we use the B module to describe a subsystem. The combination of some B modules constitutes a complex software system. The following describes how to develop the complex software systems with the B module, and the rational choice of implication relations of the invariants.

3.1 Develop complex software systems with the B module

Developers use the B method to develop the complex software systems, that is, the developer builds software systems according to the concept of the B method, and ensures the safety of the developed software systems [6]. Developers can use the B modules to describe a subsystem of the software, therefore, the combination of several B modules constitutes a complete system. Each module has its specifications and implementation, their development is independent. The B module must have an abstract machine, which is used to describe the specifications of the system. The content of the abstract machine is an objective fact and describes the system “what to do”. The B module have an implementation, which is used to describe the concrete implementation of the corresponding specification. In other words, the content of the implementation is the details and describes the system “how to do”. There may

be several transition machines between the system specification and the implementation, which are refinement machines.

Sometimes the specifications that need to be described are complex and need to be decomposed. The abstract machine can use INCLUDES, USES, and SEES to construct more complex abstract specifications, like *M* INCLUDES *N*, *M* regards *N* as a part of itself. SEES can be used to refer to information in another abstract machine. The implementation can use IMPORTS and SEES to construct more complex software systems. For example, an implementation needs to import other abstract machines as the basis. It should be noted here that IMPORTS can only be used to import abstract machine specifications, but its connection relationship can be reflected to the implementation [6].

Developers must describe the static laws of the system, that is, invariants. The invariant describes the properties that must be maintained between variables, constants, etc., the properties that must be not violated at the run-time of the operations. When the value of variables is updated, the invariant should be preserved [4]. For some developers, they may think that the relevant properties of the system need to be verified in the implementation because what is described in the implementation is the final implementation, although they know that the specification has described in the abstract machine. This is a misunderstanding.

In fact, after describing an operation in the abstract machine, developers need to ensure that this operation preserves the invariant of the abstract machine. The invariant here can not only describe the value range and mutual relationship of variables but also can be used to describe certain safety properties that must be satisfied at the run-time. So how to ensure that the concrete operations of the implementation also preserves the invariant? The developer only needs to ensure that the operations in the implementation satisfy the specifications of the abstract machine, then the concrete operations in the implementation preserve the invariant of the abstract machine at the run-time, thus ensuring the safety of the system, as shown in Fig. 1.

Without understanding how to develop complex software systems with the B module, developers may not be able to verify safety-critical properties, or even determine where to verify safety-critical properties, abstract machine or implementation?

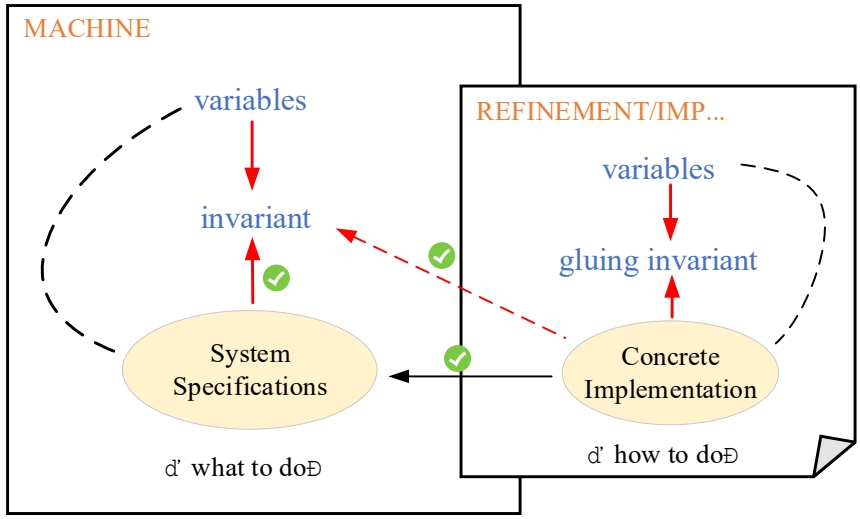

Figure 1: The B module development framework.

3.2 Implication in invariants

There are two forms of implication in invariants: $THEN \rightarrow IF$, it is consequent driven implication in this invariant; $IF \rightarrow THEN$, it is antecedent driven implication in this invariant. When the developer verifies the system properties, the appropriate form of invariant should be chosen to ensure the accuracy of implication of invariants for the verification of the safety properties.

For the rational choice of implication relations of the invariants, we conclude three following scenarios:

1. When there is only one unique transition between state A and state B, the invariants in the form of $Antecedent \rightarrow Consequent$ and $Consequent \rightarrow Antecedent$ can be both used to verify the properties of the system, as shown in the following Fig. 2(a).
2. When temporal property is involved in the *IF* statement, even if there is one unique transition between A and B, $p \rightarrow q$ may not hold, but at this time $q \rightarrow p$ holds.
3. If there are two or more transition paths between state A and state B, when $p' \rightarrow q'$ holds, $q' \rightarrow p'$ may not hold, in the case shown in Fig. 2(b).

When the form of invariant derived from the safety properties is $THEN \rightarrow IF$, the invariant may not be able to verify this safety property of the system. A case is shown in Fig. 2(c), Whether the value of the *checkPoints_pointInPosition* is TRUE or FALSE, both of these contradictory invariants are true at the same time. Why do the two invariants hold at the same time? As illustrated by Fig. 2(b), the *checkPoints_pointInPosition* is TRUE in state A', FALSE in state A", and in state B the value of *PCS_routeSectionCmd_lock* is equal to 15. But when *checkPoints_pointInPosition* is TRUE, it does not represent the state A', nor does its FALSE value represent A". Therefore, the invariant in Fig. 2(c) cannot be used to verify the properties of state transitions between A'-B, A"-B. If the description of the system states in the invariant is incomplete, verification process will produce unexpected errors because this type of invariant has no constraint on the state of the system.

When writing the invariants for the verification of safety properties, the developer should distinguish the implication of different forms invariant, then choose the appropriate and correct form of invariant for safety-critical properties verification, especially the case as shown in Fig. 2(b).

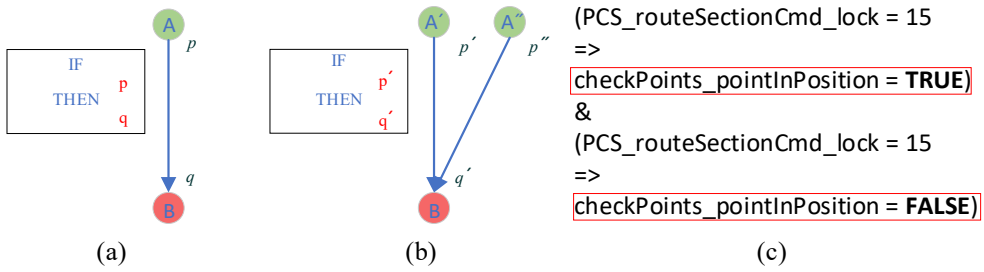

Figure 2: The implication in invariants.

4 DEFINITION OF THE VARIABLE FOR CODE GENERATION

Atelier B supports the transformation of the B models into high-level programming languages, like C codes. In this section, we focus on two forms of modelling that may be inappropriate. The C codes generated by this modelling form will cause unnecessary troublesome when used, then show how to avoid them.

4.1 The influence of the output parameters of the B model on the generated C code

Considering the following example, First, we define tt as a variable, and the value of operation $plusStart(aa,bb)$ is assigned to tt, as shown in Fig. 3(a). Next, instead of defining a variable, we only define nn as an output parameter, as shown in Fig. 3(b). The function of the operation is the same.

We found that if the operation has the output parameters, the generated C codes are difficult to understand and need to be modified (except adding the *main* function) because the output parameter nn of the C codes does not get the value, the developer needs to modify the *Function* types in C codes, thus destroying the integrity of the generated code. If the operation has no output parameters but with a variable which gets the output value of the operation, the generated C codes are more standardized and easier to use, the variable $Observer\_tt$ can get the value, this issue is shown in Fig. 4.

Developers should draw the lesson from this above case when defining the operation in the B model if they would like to get the better C codes, they may use the way shown in Fig. 3(a) to build the B model.

```
IMPLEMENTATION Observer_i
REFINES Observer
IMPORTS
   Plus
CONCRETE_VARIABLES
   tt
INITIALISATION
   tt := 0
OPERATIONS
   ObserverStart(aa,bb)=
      tt <-- plusStart(aa,bb)
END
```

(a)

```
IMPLEMENTATION Observer_i
REFINES Observer
IMPORTS
   Plus
OPERATIONS
   nn <-- ObserverStart(aa,bb)=
      nn <-- plusStart(aa,bb)
END
```

(b)

Figure 3: Two different forms of operation definition in B model.

```
/* Clause CONCRETE_VARIABLES */
static int32_t Observer__tt;
void Observer__ObserverStart(int32_t aa, int32_t bb)
{
   Plus__plusStart(aa, bb, &Observer__tt);
}
```

(a)

```
void Observer__ObserverStart(int32_t aa, int32_t bb, int32_t *nn)
{
   Plus__plusStart(aa, bb, nn);
}
```

(b)

Figure 4: C code generated from the models in Fig. 3.

4.2 Difference between VARIABLES and CONCRETE_VARIABLES in generating code

The variables used in the B method can be divided into abstract variables and concrete variables. The abstract variables are defined in the abstract machine, and the concrete variables are usually defined in the refinement and the implementation. For example, if an abstract variable defined in the abstract machine needs to be implemented, it has to be changed to a concrete variable in the refinement or the implementation. Here we only discuss the case where the variable name does not need to be changed when the variable implements.

Some developers might directly define the variable as a concrete variable in the abstract machine. When they compile the C codes generated by the B model, an error will occur. The error message is as follows: "[Error] static declaration of 'xxx' follows non-static declaration".

Finally, conclusions are summarized as follows: It causes errors in compiling C codes due to the inconsistency of the variable type in the ".h" file and the ".c" file which generated by the B model, if the variable is defined as a concrete variable directly in the abstract machine. Specifically, this variable is the external storage type in the ".h" file but the static storage type in the ".c" file. Remarkably, to make the C codes generated by the model more standardized and easier to understand, the variable should not be directly defined as a concrete variable in the abstract machine. The variable should be defined as an abstract variable firstly, then changed to as a concrete variable in the refinement or the implementation, although it is feasible to directly define as concrete variables in the abstract machine.

5 RELATED WORK

Formal methods have taken more attention in the industry in recent years, EN-50128 suggests the use of formal methods to develop systems with high safety integrity levels. However, the formal method faces various obstacles in the application process. For example, formal methods are based on mathematical logic, the verification process requires strong logical reasoning knowledge, reasoning process are complicated, reasoning methods and strategies are flexible. These reasons lead to higher learning cost of formal methods and application difficult. Moreover, the analysis of the final verification results relies on relevant experience and there lacks a formal verification platform for specific industry applications [10]. Abrial [11] mentioned that the B method is essentially used in train systems. Although this modelling method can also be used in energy, automotive and other industries, the engineer needs to change the long-established engineering method, which is almost impossible. Also, some developers believe that the use of formal methods in industrial software development means that extra effort is invested in formal modelling, and there are potential safety risks from the process of the specification to implementation, these problems can be solved by automatically generating codes from the formal model [12].

At present, the application of formal methods in the field of rail transportation is mainly concentrated in railway systems, such as train control systems and interlocking systems. Formal methods are rarely applied to the safety research of tram control system. With the development of trams, the safety of tram control systems in particular needs to be guaranteed. We use the B method to develop the tram control system to improve the quality of its software system and improve its safety. Moreover, the C codes are generated from the formal requirement model to improve the automation ability.

We have summarized the content that is likely to errors in the formal modelling process, which can help developers better use the B method correctly. Once the model does not satisfy

the expectation, it is meaningless to perform formal verification on the model. Besides, there are some interesting points when the B model generates the C codes, such as the influence of output parameters on the generated code. We consider that developers need to notice when generating C codes, although the B method does not specify which practice is wrong.

6 CONCLUSION

Based on our experience in developing tram control systems with the B method, we have summarized some of the issues and solutions that are likely to be encountered when developing the system using the B method, which should be useful to developers with less experience in this field.

In this paper, we illustrated (1) Using the B module to develop complex systems. It is different from the general method of expressing software systems, developers should understand the software system according to the abstract machine model;(2) Rationally choose the implication relation of the invariant. We summed three cases, developers can choose the appropriate invariant form by comparison; (3) The influence of output parameters on the generated code. We generated code for the B model with and without output parameters and found that the code generated by the model without output parameters is easier to understand and use; (4) The influence of directly defining concrete variables in the abstract machine on code generation. Although the B method allows concrete variables to be defined in the abstract machine, the error related this concrete variable will appear when the generated C code is compiled, and the codes must be modified before they can be used. These points are critical for developers to develop systems with the B method. These can help developers avoid unnecessary work and enhance their confidence while reducing learning costs.

ACKNOWLEDGEMENTS

This work was supported by the National Natural Science Foundation of China (No. 71502146, 61673320). We would like to thank Dr. Colin Snook for his discussion in this work.

REFERENCES

[1] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill: New York, 2005.
[2] Lecomte, T., Deharbe, D., Prun, E. & Mottin, E., Applying a formal method in industry: A 25-year trajectory. *Brazilian Symposium on Formal Methods*, pp. 70–87, 2017.
[3] Bicarregui, J.C. et al., Formal methods into practice: Case studies in the application of the B method. *IEEE Proceedings Software Engineering*, **144**(2), pp. 119–133, 1997.
[4] Abrial, J.R., *The B-Book: Assigning Programs to Meanings*, Cambridge University Press: Cambridge, 1996.
[5] Robinson, K., The B Method and the B Toolkit. *International Conference on Algebraic Methodology and Software Technology*, pp. 576–580, 1997.
[6] School of Mathematical Sciences, Peking University. www.math.pku.edu.cn/teachers/qiuzy/fm_B/slides. Accessed on: 13 Mar. 2020.
[7] Lano, K., *The B Language and Method: A Guide to Practical Formal Development*, Springer-Verlag: New York, 1996.
[8] Atelier B, www.atelierb.eu/en/atelier-b-tools/. Accessed on: 11 Feb. 2020.
[9] ProB, www3.hhu.de/stups/prob/. Accessed on: 15 Feb. 2020.

[10] Bjørner, D. & Havelund, K., 40 Years of formal methods. *International Symposium on Formal Methods*, pp. 42–61, 2014.
[11] Abrial, J.R., On B and Event-B: Principles, success and challenges. *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pp. 31–35, 2018.
[12] Mashkoor, A., Kossak, F., Biró, M. & Egyed, A., Model-driven re-engineering of a pressure sensing system: An experience report. *European Conference on Modelling Foundations and Applications*, pp. 264–278, 2018.