# Starting B Specifications from Use Cases

Thiago C. de Sousa[1] and Aryldo G. Russo Jr[2]

[1] University of São Paulo
thiago@ime.usp.br
[2] AeS Group
agrj@aes.com.br

**Abstract.** The B method is one of the most used formal methods, when reactive systems is under question, due to good support for refinement. However, obtaining the formal model from requirements is still an open issue, difficult to be tackle in any notation due the background distance between the requirement engineer and the one in charge of work with a formal specification. On the other hand, use cases have become the informal industry standard for capturing how the end user interacts with the software by detailing scenario-driven threads. Furthermore, the scenarios steps provide an easy way to derive functional tests, as in the same way what has to be achieved and what's not is, normally, clearly stated. In this paper we show how controlled use cases and functional tests based on them can be used as a guideline for writing B operations and invariants. As a side effect, we also present a practical way to establish traceability between functional requirements and formal models.
**Keywords.** B method, requirements traceability, use cases transactions

## 1 Introduction

B [1] is a formal method that allows us to produce proof obligations that demonstrate correctness of the design and the refinement. Nevertheless, there is no standard mechanism for mapping requirements to formal specifications. To overcome this issue, different solutions have been proposed by researchers. In [2], the authors have presented a traceability between KAOS requirements and B. Some authors are investigating the use of the Problem Frames approach [3] as a possible response. A mixed solution using natural language and UML-B has been proposed by [4]. However, these approaches use non-standard artifacts for requirement specifications, which we consider a disencentive for convincing designers to adopt formal methods since they must spend time to learn them.

Use cases [5] can be considered as the *de-facto* industry standard for requirement specifications. They provide a good way to capture how the end user interacts with the system by detailing scenario-driven threads. A typical use case describes a user-valued transaction in a sequence of steps expressed in a natural language, which makes use cases readable for most end-users. In [6] the author has presented an approach for building B specifications from use-case models. This approach is similar to ours, but his project has focused on bringing the object-oriented paradigm (including UML diagrams) to formal methods. His method also maps each use case as a unique B operation, what we believe is not correct since each use case can have many transactions according to Jacobson[7].

Another relevant point about uses cases is the possibility to derive test cases. One of the new proeminent development model is the so-called Test Driven Development[8], where the input information used to generate the source code are the test cases, instead of use case scenarios or other traditional requirement documentation.

In this paper we propose an approach for starting B specifications from use and test cases. Use cases transaction identification can be used as a guideline for defining B operations, including the pre- and postconditions. In the same way, test cases can help on the definition of global invariants and constraints. This research is part of a bigger project, called *BeVelopment*[9], partially supported by DEPLOY project. This project is presented in the section 2.

In the section 3 we explain our approach in more details showing how it works on an example for booking flights for traditional B and a small (industrial) example used to explain our approach for Event B. In the section 4 we introduce a possible utilization of annotation techniques to facilitate the process. This annotation process can be viewed as a refinement in a informal way, were constraints and supported information are added to contribute for a better understanding of the problem, and to help in the formal refinement steps.

Finally, we reserve the last section for further discussions and enumerate some of the future works.

## 2 BeVelopment Project

As can be seen in figure 1, the development process is composed by several phases and each phase composed by several tasks. This model was extracted from the IEC 61508[10] standard, although it's similar in several different fields of application.

Most of the time, when this model is followed, in safety critical applications, formal methods are used only in a small part of the process. Our objective here is to spread formal method utilization in almost all the phases in this process, and ultimately, create a guide of application that could be used by others in order to introduce this extended methodology.

For that, we intend to use what's been developed inside DEPLOY workpackages (see figure 1 for details) to create a chain of application.

We expect to be able to integrate what's been developed in Workpackage 1 and 4 during the system and software specification phase. The effort here will be in determine a way to specify better requirements to avoid that errors like ambiguities and inconsistencies move forward to later phases. There are also other studies we'd like to investigate in this phase like in [11] and [12]

During the software architecture and design we intend to use what's been developed in workpackage 2 and 3 along with the decomposition technique that's on development, trying with that, in the determined point separate hardware part and software part, and from that continue with the next refinements independently from each other.

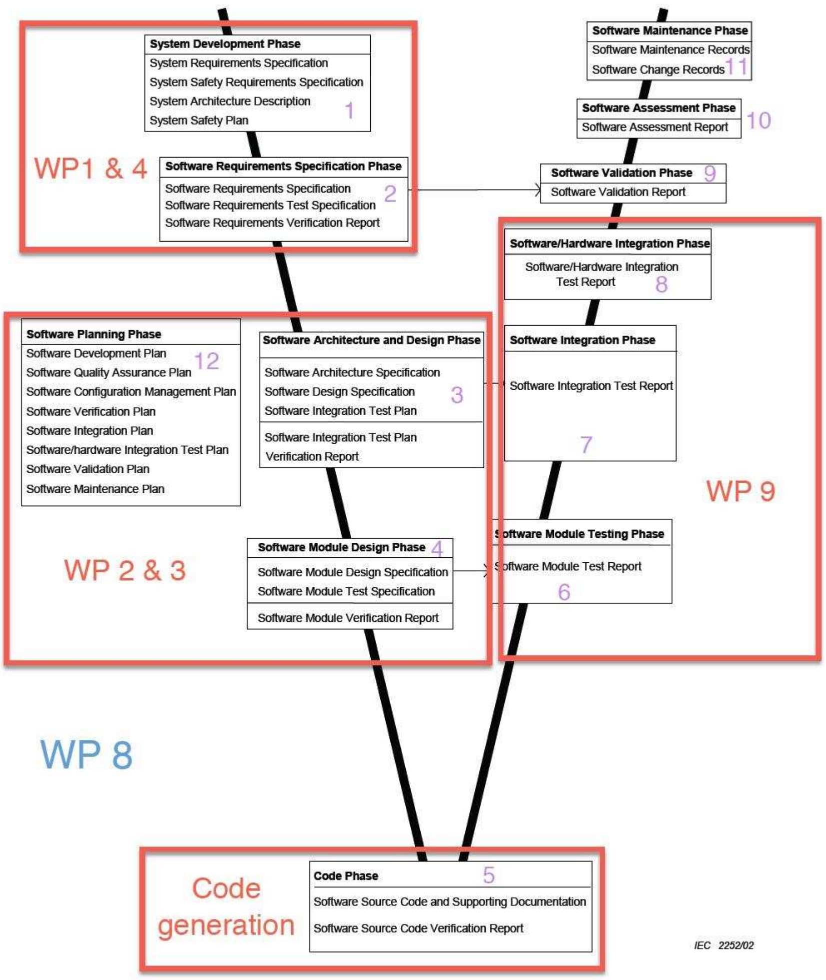
System Development Phase
System Requirements Specification
System Safety Requirements Specification
System Architecture Description
System Safety Plan
1
WP1 & 4
Software Requirements Specification Phase
Software Requirements Specification
Software Requirements Test Specification
Software Requirements Verification Report
2
Software Planning Phase
Software Development Plan
Software Quality Assurance Plan
Software Configuration Management Plan
Software Verification Plan
Software Integration Plan
Software/hardware Integration Test Plan
Software Validation Plan
Software Maintenance Plan
12
Software Architecture and Design Phase
Software Architecture Specification
Software Design Specification
Software Integration Test Plan
Software Integration Test Plan
Verification Report
3
Software Module Design Phase
Software Module Design Specification
Software Module Test Specification
Software Module Verification Report
4
WP 2 & 3
WP 8
Code generation
Code Phase
Software Source Code and Supporting Documentation
Software Source Code Verification Report
5
Software Maintenance Phase
Software Maintenance Records
Software Change Records
11
Software Assessment Phase
Software Assessment Report
10
Software Validation Phase
Software Validation Report
9
Software/Hardware Integration Phase
Software/Hardware Integration Test Report
8
Software Integration Phase
Software Integration Test Report
7
WP 9
Software Module Testing Phase
Software Module Test Report
6
IEC 2252/02

**Fig. 1.** V model and Workpackages

After code generation phase, that we hope it would be possible to be done automatically from the refined specification, we intend to use what's been developed in workpackage 9, in the sense of helping the tool development group in create an appropriate plug-in to extract test cases from the formal specification.

At the end, the whole process is part of a more wide objective that is improve the dependability of the developed product, which meets the expectation of the workpackage 8.

### 2.1 Project details

The main objective of this project is to create a useful methodology to be used during the development life cycle of safety critical systems. In order to do that, is a fact that some ingredients are needed, as follow:

1. A development life cycle *framework*. This framework must define what are the phases in this life cycle, and what are the inputs and outputs of these phases. Moreover, it needs to state what are the tasks that need to be performed to "transform" the inputs in the correspondent outputs of each phase. There are several frameworks that could be used, like spiral, clean room, XP, etc.. but, as it is the case of railway domain, the V model would be the one used in this project.
2. For the "transformation", or to perform each task, it's necessary some *techniques* (or languages, tools, etc...) that would be used to get the inputs and generate the expected outputs. As the objective of this project is the application of formal methods (in our understating, formal methods are, in fact, formal languages by the fact of lack of a utilization method, like is stated in [13]) during the development life cycle, one of the expected results is the identification of what method and related tools would be suitable. As in some other previous studies, we have strong feelings that the B method and its derivatives would suit well in most of the cases in railway domain. But, where is the case of necessity other formalisms would be applied, and as a secondary objective in this project we would like to evaluate, compare, and verify other methods like VDM [14], Z [15], and others.
3. But, to be able to use such techniques, an utilization (or application) method need to be used in order to guide, or to state the steps that are necessary to successfully achieve the objectives. In almost all cases, such formal languages are not followed with this methods, and, as was stated by Jens Bendisposto and Michael Leuschel, during Dagstuhl Seminar (Refinement Based Methods for the Construction of Dependable Systems), using the example of the "Abrial index", where can be seen that when these formal languages are used by people that really knows about that quite well, the resulting specification is easily proved where is not the case when it's done by people who not follow a (hidden) method. During this project, as another secondary objective, we would like to determine a method that would guide these techniques application, to help people during the development life cycle to spend their time in valuable tasks, and not in "try and error" experiences.
4. Finally, as the *methodology* itself, is the task to determine the transitions from one phase to another. It needs to be a guide that state what intermediate tasks are needed in order to an output of a previous phase could be used as input of the next one. Moreover, it's the translation of the used framework

in words that state how to perform whatever is needed to achieve the end of the life cycle, and not only the "what" needs to be done. This is the main objective of this project, and we hope it could be generic enough that could be used in other domains, but strong enough to facilitate the adoption of formal methods in railway domain as a strong methodology that helps the accomplishment of what is already required by the domain standards.

The work presented in this paper is related to the system analysis and physical model, presented in 1. At this stage it's not our aim an automatic translation (although it's likely to be a necessity in the near future) from the use cases to the formal model, instead, we intend to facilitate the manual process.

## 3 Mapping UC to B

The aim of this approach is to fulfill some intermediate phases in the proposed development process. Until now, it's commonly seen the use of formal methods in the development process, only from the design phase where the requirements are already well defined, but this is not the case on industrial projects.

Based on that, the method proposed here would be useful to be used during early phases, to help on both directions:

– to the top, helping the requirements elicitation (what might be combined with other techniques and methods, like Jackson's Problem Frames)
– to the bottom, helping the creation of the first abstract formal model, in the sense that it can support the first definitions.

For mapping use cases to traditional B specifications we propose that use case scenario sentences must be written using a controlled natural language (CNL) described according our use case transaction definition, which is based on Ochodek's transaction model [16].

**Definition 1.** *A transaction is a shortest sequence of actor's and system actions, which starts from the actors request and finishes with the system response. The system validation and system expletive actions must also occur within the starting and ending action. The pattern for a transaction written as a sequence of four steps in a scenario:*

*n. An actors request action (U).*
*n+1. A system data validation action (SV).*
*n+2. A system expletive action (e.g. system state change action) (SE).*
*n+3. A system response action (SR).*

We have also decided to define the grammar using subject-verb-object (SVO) sentences because they are good at telling the sequence of events. We have mapped the use case actor as subject, a set of actions predicate synonyms (for example validate, verify etc. would be grouped together) as verb and the rest of the sentence as object.

For the sake of simplicity, we assume that all environmental tasks, like user interactions, are perfect, i.e., they are executed always in a correct order and in a correct moment. Further studies will be developed to incorporate non-functional requirements and exceptions.

Let's take a look at an example in order to clarify our idea, first about the generation of controlled use cases, then we introduce the annotation and finally, the test cases generation. In accordance with previous definitions, suppose we have the following use case scenario:

```
1. The agent specifies a travel itinerary for a client.
2. The system validates the itinerary.
3. The system searches a set of appropriate flights.
4. The system presents them to the agent.
5. The agent selects a flight.
6. The system verifies free spaces on the flight.
7. The system reserves any seat from the set of free spaces.
8. The system confirms the reservation.
```

In the above example, we can see two transactions (1-4 and 5-8) and each one can be used as a guide to create a B operation. From SV actions (2 and 6) we extract the preconditions (validates the itinerary, verifies free spaces) and from SE actions (3 and 7) we derive the operations names (search, reserve) and the postconditions (searches a set of flights, reserves any seat). A possible mapping to B specifications, which is self-explanatory, is shown below.

```
flights ← search(city1,city2) ≜
    PRE city1 ∈ CITY ∧ city2 ∈ CITY ∧ city2 ∈ itinerary(city1) (2)
    THEN flights := companies({city1 ↦ city2}) (3)
    END;

reserve(flight) ≜
    PRE card(flight) > 0 (6)
    THEN ANY seat WHERE seat ∈ flight (7)
        THEN flight := flight - {seat}
        END
    END
```

For mapping use cases to Event B specifications we must include one more step in the previous definition.

**Definition 2.** *A transaction is a shortest sequence of actor's and system actions, which starts from the actors request and finishes with the system response. The system guard recognition, validation and expletive actions must also occur within the starting and ending action. The pattern for a transaction written as a sequence of four steps in a scenario:*

*n. An actors request action (U).*
*n+1. A system guard recognition action (SG)*
*n+2. A system data validation action (SV).*
*n+3. A system expletive action (e.g. system state change action) (SE).*
*n+4. A system response action (SR).*

Another example, to explore the abstraction of the first model for Event B, can be seen bellow. This is an example based on a train door system, where, when a open command is received by the system, if the conditions are satisfied, the train door must open. This is an attempt to use this approach at an abstraction level, but it seems to be strong enough to present the general information of the system.[3]

```
1. The agent requests to open the doors on one side of the train.
2. The system recognise the request.
3. The system verify the validity of this request.
4. The system commands the opening.
5. The system confirms the execution.
```

We can use SG, SV and SE actions as a guide to derive guards, conditions and actions. A possible mapping for an Event B model was created from this use case:

```
OPEN_COMMAND ≜
    WHEN grd1: REQUEST = TRUE (2)
    WITH cond: CONDITIONS = TRUE (3)
    THEN act1: OPEN := TRUE (4)
    END
END
```

At this point we just present superficial information, complete enough to understand the global functionality of the system. Again, at this point there is no need to present:

– how the system recognize the request;
– how it must to be validate;
– how the command is generated.

These suppressed information help to understand the main goal of the system, and there is no need to express the internal behavior of the controller itself. In the next section we present a possible approach to introduce such information.

## 4 Informal UC Refinement

The second step(under development), in order to generate a full B specification, is the generation of test cases, based on use cases specification.

[3] as this is highly an abstract representation of the (one function of) the system, a lot of manual work has to be performed, like variable names, etc...

As stated before, we also intend to use test cases to help on the development of formal specification, but before that, some test cases classification is needed to help on understanding the proposition.

One possible classification of tests are:

– *test to succeed* - these are tests that can validate that the system is performing it's actions accordly with what was specified for the system to do, meaning that, based on the correct inputs, the system must provide the correct output.
– *test to fail* - these are tests that are used to verify the behavior of the system when inputs are provided in a different sequence, order, or value. It's used to check if in those cases the system has the correct protection to avoid misunderstandings or dagerous behaviors.

Based on this classification, it's not difficult to see that the generation of the first one is almost a straightforward categorization of the use cases, but the second one is not trivial.

To help the creation of the second family of test cases, we propose an annotation technique that must be applied to the use cases in order to facilitate the definition of bounderies, types, and any other information that could determine possible exceptions.

as stated before, from one single use case, it's possible to derive several test cases, divided in test to succeed, and test to fail. to help this generation, we propose a simple, informal annotation system that needs to be included in the use case specification, do drive the user to determine what test cases are necessary. Basically, this annotation system cover the following properties:

– Boundaries
– Possible values
– Safety properties (like, not allowed sequences, wrong events, etc...)

An example for this annotation (based on the first example) can be seen bellow:

```
   1. The agent specifies a travel itinerary for a client.
[AN1:-> the source and destination must exists.
 AN2:-> the valid characters are only alphabetical]
   2. The system validates the itinerary.
```

The generation of the test case related to test to succeed are straightforward, so it will not be cover here, but to generate the test cases related to test to fail, the included information (the developer/client wishes), can help a lot.

One example of test to fail based on this annotaded use case can be:

```
   1. The agent specifies, as a travel itinerary, the follow data:
   Paris12.
   2. The system verify the validity of this itinerary.
   3. The system return the related error message.
(as the property, in the annotation part, do not hold)
```

More formally, the refinement process is based on the horizontal refinement techniques [17], where new information is introduced in each step. Of course, these refinements can not be proved as correct in the Use Cases phase, but it will be verified during the formal development. Moreover, it's a complimentary tasks that needs to be reviewed in case of failures during the verification.

Based on the second example, we can have the first refinement, based on annotations, like the one presented bellow:

```
   1. The agent requests to open the doors on one side of the train.
[AN1 - this operation is made by selection
the side and pressing the correspondent bottom]
   2. The system recognise the request.
[AN2 - this operation is only allowed if the train
is stopped in the correct position(at the station)]
   3. The system verify the validity of this request.
[AN3 - all the conditions must be true at the same time]
   4. The system command the opening.
[AN4 - at least 2 different signals must be
activated to guarantee the safety condition]
```

From this example, we can see that's possible derive several test cases. Some examples are presented bellow. Again, first we present the test to succeed, and after some possibilities of test to fail. One test to succeed would be:

```
   1. The agent select the right side for opening
and press the right open bottom.
   2. The system verify the sequence and the consistency,
i.e., if the side and bottom are related,
and if the speed is 0km/k
   3. The system activate both output signals
```

This procedures can be done in two different flavors. In one side we can work on both, informal (use and test cases) and formal model at the same time, I mean, at the moment that one refinement is made in the informal model, it must be reflected in the formal one. This approach helps the mistakes discovery during the early development phases, although it brakes the reasoning flow during the construction of informal model. On the other hand, we can refine completely the informal model, and after generate the formal one. As it can help the reasoning flow, it can be a problem if errors are discovered later in the process. Based on these information, a weighting must be performed to determine when and how go to formal model and come back.

The last step, the generation of global invariants and constraints is still under study and it's supposed to be presented in a future revision of this work.

## 5 Discussions and Future Work

In this paper we have proposed an approach for mapping requirements to B (Event B) models through use and test cases in a pragmatic way. We are not interested (at this moment) in the automatic translation of use cases for formal specifications since there are many natural language ambiguity problems. The intention is to take the use cases as a guideline for starting B specifications.

Our main goal is to create a new and complete development process (including deliverables artifacts), namely *BeVelopment*, for B focusing on agility/usability and we believe that use cases seem to be a good start point.

Other approaches are under study right now like Problem frames and KAOS. The intention is to facilitate as much as we can the beginning of the development process where formal methods are supposed to be used.

The project is in the initial phase and there are a lot of future works. First, we are planning to use the Rodin platform as tool support. Another possible improvement would be a more flexible grammar since there are B operations without preconditions. Alternative scenarios (extensions) must also be included in the transaction discovery process. At the moment, our approach only maps B operations (events for Event B) and we are investigating other artifacts for extracting the B machine name, constants, properties, assertions, variables, initializations and invariants as well as non-functional requirements, including performance and reliability.

## 6 Acknowledgments

We'd like to thank to Prof. Ana C. V. Melo, who devoted his time to review this paper and mostly for her comments which help us to include substantial points in this work.

## References

1. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York, NY, USA (1996)
2. Ponsard, C., Dieul, E.: From requirements models to formal specifications in B. ReMo2V CEUR Workshop Proceedings (2006)
3. Jackson, M.: Problem Frames: analyzing and structuring software development problems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
4. Jastram, M., Leuschel, M., Bendisposto, J., Jr, A.R.: Mapping requirements to B models. DEPLOY Deliverable - Unpublished manuscript (2009)
5. Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley Professional (1992)
6. Ledang, H.: Automatic translation from UML specifications to B. Automated Software Engineering Conference (2001)
7. Jacobson, I.: Object-oriented development in an industrial environment. In: OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications, New York, NY, USA, ACM (1987)
8. Beck, K.: Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
9. Jr, A.R.: DA associate program - AeS proposal. Unpublished manuscript (2009)

10. Bell, R.: Introduction to IEC 61508. In: SCS '05: Proceedings of the 10th Australian workshop on Safety critical systems and software, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2006) 3–12
11. Lamsweerde, A.v.: Goal-oriented requirements engineering: a guided tour. Proceedings of Fifth IEEE International Requirements Engineering Symposium (2001) 249–262
12. Gunter, C.A., Gunter, E.L., Jackson, M., Zave, P.: A reference model for requirements and specifications. IEEE SOFTWARE **17**(3) (2000) 37–43
13. Mazzara, M.: Deriving specifications of dependable systems: toward a method. In: Proceedings of the 12th European Workshop on Dependable Computing (EWDC). (2009)
14. Bjørner, D., Jones, C.: The Vienna development method: The meta-language. Springer-Verlag, London, UK (1978)
15. Woodcock, J., Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)
16. Ochodek, M., Nawrocki, J.: Automatic transactions identification in use cases. In: Balancing Agility and Formalism in Software Engineering, Berlin, Heidelberg, Springer-Verlag (2008) 55–68
17. Abrial, J.R.: Faultless systems: Yes we can! Computer **42**(9) (2009) 30–36