Formalization of Requirements for Correct Systems

Imen Sayar, Jeanine Souquières

Formalization of Requirements for Correct Systems

Imen Sayar
University of Luxembourg
L-1359 Luxembourg
imen.sayar@uni.lu

Jeanine Souquières
University of Lorraine, CNRS, LORIA
F-54000 Nancy, France
jeanine.souquieres@loria.fr

***Abstract*—Improving the quality of a system begins by their requirements elicitation: the challenge is to bridge the gap between the requirements of the client and their formal specification defined by the scientist. A first step consists on understanding and rewriting the existing requirements. Along the development process, we introduce formal terms in the requirements coming the formal specification and make explicit the interactions between them by a glossary. The trace of the requirements and their corresponding specification is managed and serves to simplify the activities of validation and verification. The validation is studied since the understanding of the first requirements and all along the development of their formal specification. The verification may detect imperfections like incoherences and ambiguities in both the formal specification and their corresponding requirements.**

***Index Terms*—Formal specification, glossary, refinement, requirements, tool, traceability, validation, verification**

I. Introduction

A requirements document serves as a bridge between the clients and the suppliers of software and systems development. This document is used as a binding agreement and needs to be understandable by both the involved parties. It is ironic that the natural language, used to facilitate the communication between the clients and the developers, introduces unwanted ambiguities in the requirements and misunderstandings between the stakeholders [31]. Although some rules of thumb [27] allow a well-defined requirements document, it remains difficult to get a holistic view of the system under development.

The development of formal specifications is an activity based on the cognitive skills of the person in charge of developing them out of the informally described user requirements. The purpose of modelling is not only to write a specification, it also serves to improve the understanding of the system being modelled: the requirements documents have to be understandable without ambiguity and support for the development of formal specifications. Approaches that propose the use of controlled natural language for the description of requirements, improve the requirements clarity, but do not contribute directly to the development of formal specifications. In fact, the situation has not evolved much since last decade, *"Constructive methods for building correct specifications for complex systems in a safe, systematic, incremental way are by and large non-existent"* [29].

The refinement is fundamental to deal with the complexity; a specification is constructed gradually, i.e. it cannot be defined in a single step, but it is defined from an initial abstract view to a final concrete one. This notion, introduced in the beginning of the 1970s [8], is a technique of transforming an abstract model into a concrete one. The latter must preserve the properties of the abstract model as well as its behavior. The refinement of the formal specifications allows the incremental development of a complex specification starting from an abstract model and concretizing it step-by-step by introducing more concrete details into the formal model. The view of the system gets more accurate. The concrete model must be consistent with the abstract model. The consistency of this formal model is ensured via proof of refinement.

The validation focuses on responding to the question, "are we developing the correct system ?", checking if the developed formal specification answers the requirements of the client. It is almost interested on the behavioural aspect of the specification. The verification aims to answer "are we constructing the system correctly ?" and concerns the formal specification proving its mathematical correctness using tools like provers and proof obligations generators.

Contribution. Models cannot be developed with pen and papers because they are too big and complex. Our development is a mix of manual and semi-automatic steps. A manual task depends on the developer skills who build the system by introducing details in the specification and rewriting the client requirements. Our development patterns formalize requirements and semi-automate the development of parts of the formal specification.

In our development, we have developed Event-B [3] and TLA^+ [16] specifications. In this paper, we use Event-B, formal language for modeling and reasoning about large reactive and distributed systems. It is based on set theory and standard first-order predicate logic; the models are developed through stepwise refinements. An Event-B model is described using contexts and machines. A context contains sets and constants having properties defined in predicates called axioms. A machine has a state characterized by variables and constrained by invariants. The transition between machine states is ensured by events containing guards and actions. We use the Rodin Platform[1] and different existing plug-ins. Our contributions concern :

- The validation. It begins since the understanding of the requirements and goes on all along the development of

[1] http://www.event-b.org/

their formal specification. We deal with the validation using two aspects, static and dynamic. These aspects are fulfilled using animation, model-checking and definition of validation scenarios which make the specification more trustworthy.
- The history of abstract specifications memorised in order to reuse it for the refined specifications. It helps the validation and takes into account the glue between different levels of the development.
- The use of existing tools. ProR supports requirements engineering. The verification and validation (with ProB and JeB) tools allow to uncover incoherences, omissions and imprecisions in both requirements and their specification.

The remainder of the paper is structured as follows. Section II presents our approach, using development strategies and tools like validation and verification. Section III describes the validation all through the development process, starting from the understanding of initial requirements. Related works are discussed in Section IV and finally, Section V concludes the paper.

II. Approach

We start from the requirements document describing informal and semi-formal artifacts of user's point-of-view of the system under development [12].

A. System state

A system is defined by a sequence of development steps, where a step is composed by the requirements, the specification and a glossary.

A.1. Requirements: Informal when we start the development, they are rewritten into short and simple sentences. Their trace is saved, following the development. The resulting document is defined by a sequence of hierarchical typed sentences. A sentence is a cartesian product of several fields. The development evolves and formal terms coming from the *Specification* are introduced in the *Requirements*, using ProR for structuring and tracking requirements

ID	*Description*	*Terms*	*Term_Type*
...	...	...	...

where
- *ID* defines a unique identifier of each requirement.
- *Description* contains an initially informal description of the requirements. This will include formal terms as the formal specification evolves.
- *Terms* refers to a set of terms extracted from the sentences of the *Requirements*. They are initially informal and become formal when they are taken into account in the *Specification* and put between brackets [ ] in the *Requirements*.
- *Term_Type* describes the category of the terms included in the sentences. Based on Event-B components, we define different categories as detailed in the next sub-section.

NB. Our approach is not dependent on a specific tool, however we use the ProR tool [14] for documenting the requirements.

A.2. Specification: When using Event-B, the specification is defined by the following parameters:
- *ID*. It is a comment.
- *Terms*. They correspond to the names of context, machine, carrier sets, constants, variables and events.
- *Term_Type*. The type of a term corresponds to:
  - *Fact* for the constants, sets, variables and events parameters,
  - *Actor* for the name of a machine and context,
  - *Functionality* for the events including the actions,
  - *Obligation* for the axioms, invariants, theorems and guards,
  - *Behavior* for the animation and simulation of the specification using sequences of events and
  - *Hypothesis* for the external constraints. They are supposed to be taken into account.

A.3. Glossary: It denotes the link between a formal term of the specification and its corresponding informal definition in the requirements. Initially, the specification is empty and the requirements are informal. Throughout the development, the formal terms introduced in the specification are automatically integrated in the requirements denoted between [ ]. The *Glossary*, managed by the ProR tool, is automatically updated and always available. It is composed of couples of terms (formal term, informal description) in which the first comes from the specification and the second describes its informal definition in the requirements. It allows to detect when different names concern an informal element or a formal name is linked to different informal elements.

B. Development strategy

The construction of specifications is often a combination of smaller sub-components. Composition and decomposition techniques [24] allow formal combination of sub-components through refinement steps. A top-down approach starts with a general picture of the system, breaking down it to gain insight into its compositional sub-systems. By convention, the bottom-up approach starts with details and structures the system through the interaction of the sub-systems seen as independent systems.

- *Decomposition.* In a top-down approach, the development process starts with a general view of the system state. It is decomposed into different sub-models which can be refined independently. The complexity is reduced by the size of models and the modularity of large systems is increased. This approach allows the refinement of components in parallel. We have two types of decomposition in Event-B: the shared-variable decomposition for modelling parallel systems and the shared-event decomposition [25] for modelling messaging-passing distributed system. A Decomposition plug-in of

Rodin allows to decompose a model into sub-models.

- *Composition.* In a bottom-up approach, the development starts with requirements concerning different individual components. The global system is a combination of smaller sub-components by reusing and combining them through refinement steps; a gluing invariant is used to relate abstract and concrete variables. Its behavior is defined from interactions among them and between components and the environment. Proof obligations ensure valid compositions and shared event composition preserves refinement proofs: proving the refinement between corresponding sub-components maintains the refinement of the composition. A Shared Event Composition Plug-in of Rodin allows to compose different models [24].

- *Development patterns.* We have defined some development patterns linked to the shape of the requirement [23]. For example, *dev-if* operates on conditional forms and generates a part of the specification under development, the informal requirements constraints are automatically introduced in the existing system and the correctness of the specification has been proved once and for all. This development pattern offers elements related to the validation activity. These elements are generic and automatically instantiated.

We have defined a *Machines-compose* development pattern operating on the composition of two existing machines *M1* (with its events *e1* and *e2*) and *M2* (with its event *e3* and *e4*). The resulting system is described by:

- *Requirements.* The glue between the existing machines *M1* and *M2* has to be defined.
- *Specification.* The definition of the global machine *M* is the result of the shared event composition of the machines *M1* and *M2*. The two events *e2* and *e3* are composed. A composed event is introduced to describe the interaction between *M1* and *M2* by a gluing event.

C. Tools

C.1. Managing requirements: We use *ProR* to edit requirements and link them with formal specifications, using a glossary. Requirements can be organized in a hierarchical structure following:

- a *brother-brother* relationship where a requirement is in the same level as a *"brother"* one and
- a *parent-child* relationship where a *"child"* requirement details a *"parent"* one.

C.2. Validation: We use ProB for the animation and the model-checking of an Event-B specification [17]. The simulation using JeB allows the automatic validation of behaviors. In [22], we have proposed the validation of the formal specification relatively to its requirements, using the glossary, representing the links between these two documents, see Figure 1.

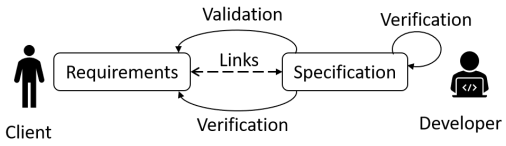

Fig. 1. Verification and validation activities

The validation starts from the phase of the analysis of the requirements during which we extract elements from the client requirements. A validation element refers to terms according to which the specification will be validated. These terms concern:

- the *fact* and *actors* presented in the system,
- the expected *functionalities* and services provided by the system,
- the *conditions* and *obligations* under which the system works,
- the defined *behavior* using existing functionalities, described as a sequence of operations or services and
- the *hypothesis* explaining a phenomenon.

Using *ProR*, we introduce new fields in the requirements containing informations extracted from each sentence:

ID	*Terms*	*Term_Type*	*Event-B Model*
...	...	...	...

These informations concern validation elements and represent formal and informal terms according to which the specification should be validated. They are classified into the previously mentioned categories.

C.3. Verification: Each machine and context under development must be mathematically correct. This correction is achieved through the proof obligations noted POs. A PO is a predicate that defines a property which must be proven for the formal Event-B model. The POs are generated automatically by the proof obligation generators in the form of sequents. For a goal G and a set of hypotheses H, a sequent of the form $H \vdash G$ means that G must be demonstrated using H. The POs are subsequently demonstrated by the provers such as the SMT [7] and SAT [9] solvers. These provers are composed of a set of "reasoners" written in Java. Each reasoner takes as input a sequent, it verifies it, and if it succeeds, it provides a proof rule. The latter corresponds to a proofs tree generated by a tool called sequents calculator. A proofs tree is associated with each PO showing the path used to prove it.

A successful PO is called *discharged*; otherwise, it is failed. Even with the existence of powerful proof tools, the location of the source of the POs failure remains a difficult task to accomplish and requires significant efforts of the specifier. In order to handle the failed proofs, a new attempt of proving

consists on either the intervention of the specifier within the framework by mean of interactive proofs, or the relaunch of automatic provers. The proving activity [4] focuses on three fundamental properties:

- the correct definition of expressions,
- the preservation of the invariant and
- the correction of the refinement.

Several proof obligations emerge from the next properties:

PO	*Description*
INV	The invariant is preserved by the machine events.
GRD	The guards of abstract events are reinforced in the refined events.
THM	Every theorem is provable.
WD	The axioms, invariants, theorems, guards and actions are well defined.

The feedbacks of this activity give indications about shortcomings in the requirements document like contradictions or oversights [2]. In [1], the author lists four outputs of the proving activity: (1) the specification is correct, (2) the specification is not correct, (3) the proof is failed but the specification may be probable or not, an (4) the proof fails but the specification may be neither provable nor refutable. The third and the fourth cases reveal that either the specification and the requirements should be reviewed, see Figure 1.

The non-discharged POs give an indication of a problem and its origin. This indication leads to a revision of the specification and/or the requirements document. It concerns the detection of the imperfections in the requirements using the verification activity. In fact, the origin of the specification concerned by the verification is the requirements document. This document may contain inconsistencies and anomalies that lead to failure of the proofs.

C.4. Other tools: We use the following tools of the Rodin platform, offering a multi-vision of the different components.

- A Project Diagram visual representation[2]. It includes Event-B root components such as Machines and Contexts, and relationships between them. It helps users to better understand complex models.
- The Event-B Statemachines tool[3]. It provides a way of adding state machines directly in to Event-B machines. The statemachines generate additional guards and actions added to the events existing in the same machine. This tool offers another vision of Event-B machines under construction and allows to see the changes of their states following the triggering of one or more events.
- The composition and the decomposition of Event-B models. They are plans to support two styles of composition in which sub-models interact via shared variables or synchronisation over events, with the Rodin support [25]. The proof obligation generator is extended to enable independent refinement of sub-models.

[2]http://wiki.event-b.org/index.php/Project_Diagram

[3]http://wiki.event-b.org/index.php/Event-B_Statemachines

III. Validation along the development

We tackle the validation all through the development process. We firstly prepare this activity while rewriting the existing requirements and validate the specification right from its starting development and all along its progression. For this, we take into account the category of the terms included in the requirements sentences.

A. Understanding the requirements

While writing elementary requirements, a preparation for the validation activity of the future Event-B model is gradually introduced. Such action aims to take into account this activity as a rigorous process along the formal development process, and from its upstream phases. We apply two steps when rewriting elementary requirements, using the tool ProR.

A.1. Classify each requirement.: It is based on the choice of the formal language. This classification is mentioned in the rewritten requirements document and the following kinds are distinguished.

- *Fact* describes the requirements containing data that should be present in the future software.
- *Actor* refers to *" a person, a system or a technical device in the context of a system that interacts with the system "* [11].
- *Functionality* contains the requirements describing the future functionalities or services of the system.
- *Obligation* describes the requirements presenting constraints, expressed in terms of *precondition* and *postcondition* of Hoare triple[4], under which the functionalities of the future system must work.
- *Behavior* contains the requirements describing a sequence of services allowed or not in the future system.
- *Hypothesis* defines the assumptions under which the system works [13]. This type is trivially taken into account by the system.

A.2. Extract validation elements.: According to these elements, we will validate the future Event-B model. They are extracted from elementary requirements, according to their kind(s). To do that, we ask respectively the questions:

- Which *facts* should be present in the future system?
- Which *functionalities* are expected in the future system?
- Under which *conditions* the future system must work?
- How the future system will *behave* in normal and abnormal ways?

Example. Let's us look back to the informal requirement *FUN*, which comes from the hemodialysis requirements document [18] in which three informal elements have been introduced:

[4]https://en.wikipedia.org/wiki/Hoare_logic

ID	*Description*	*Terms*	*Term_Type*
FUN	Hemodialysis treats the kidney failure	hemodialysis	Actor
		treats	Functionality
		the kidney failure	Fact

B. Validating the static and dynamic aspects

B.1.: The contexts specify the *static* part of a model. Its role is to isolate the parameters of a formal model and their properties, which are assumed to hold for all instances.

- A *Fact* is a data often described by the name of an object in an informal text of the concerned requirement. In the Event-B specification, an element of this type corresponds to a set, a constant, a variable or an event parameter.
- An *Actor* corresponds to the name of a context or machine.
- A *Functionality* corresponds to the name of events.

The validation of an element of these types ensures that the data and services of the future system are taken into account and present in the development.

Example. The *"hemodialysis"* actor coming from *FUN* is formalized as a name of an Event-B machine, *Hemodialysis_Mch*. The *"treats"* functionality is translated into an event named *"treats_k_failure"*. The fact *"the kidney failure"* becomes a boolean variable *"k_failure"*:

Formal term	*Informal description*
Hemodialysis_Mch	hemodialysis
treats_k_failure	treats
k_failure	the kidney failure

The validation of these static aspects is successfully fulfilled.

B.2.: The machines specify the *dynamic* part of a system. They encapsulates a transition system with the state specified by a set of variables and transitions modelled by a set of guarded events. We should have a system describing validation elements of these three categories:

- The validation according to *Obligations* ensures that the specification describe explicitly these obligations as invariants or guards in the events.
- The validation relatively to *Behaviors* shows that the system respects the sequence of actions described in the specification.
- The validation according to *Hypothesis* requirements describes in concrete terms what we expect will happen in the study. Hypothesis are defined by axioms.

C. Validating using a development pattern

We use the development *Dev-if* pattern, generating automatically a part of the specification under development. In [23], we have described this pattern as well as its generic extracted validation elements.

Example. We apply it on the R-5' requirement which contains formal terms (it is a rewriting of the R-5 software requirements [18]):

ID	*Description*
R-5	During initiation, if the software detects that the pressure at the VP transducer exceeds the upper pressure limit, then the software shall stop the BP and execute an alarm signal.
R-5'	[initiat] if [vp] exceeds [upper_press_limit] then stop [BP] and execute [ALM_excess_vp]

The following validation elements are automatically instantiated. The associated specification is automatically validated according to them. The understanding of the requirements is important to decide whether a fact becomes a constant or a variable. For instance, in R-5, there is a fact "the upper pressure limit" which describes a precise value to not exceed. That means that this fact will be translated into a constant. However, in the same requirement R-5 the fact "the pressure at the VP transducer" defines a value which varies and may exceed "the upper pressure limit". This fact is considered as a variable. The instantiated elements and their types in the specification are:

Instantiated element	*Type in the specification*
initiat	constant
vp	variable
upper_press_limit	constant
BP	variable
ALM_excess_vp	constant

The glossary shows the correspondence between these elements and their informal description in the informal requirement R-5:

Formal term	*Informal description*
initiat	During initiation
vp	the pressure at the VP transducer
upper_press_limit	the upper pressure limit
BP	the BP
ALM_excess_vp	an alarm signal

D. Validating scenarios

D.1.: The defined scenarios in the client requirements document are not sufficient to validate the specification.

Example. The R-5 requirement implicitly defines a bahavior that is validated using the sequence of events (they have been defined in Event-B, not presented in this paper):

start_BP → increase_vp → manage_vp_excess

The developed specification respects this scenario and is then valid according to this behavior.

D.2.: We use different diagrams and methods to define new ones such as Event-B Statemachines tool, ProB and a mutation testing technique to generate new non-allowed scenarios. *"A mutation test aims at detecting all the faults that are equivalent to mutants generated by a set of mutation operators"* [15]. A mutation operator consists on introducing errors by software developers in the existing program. The resulting program is called *mutant*. We introduce errors in existing scenarios to obtain *mutant scenarios* and animate the specification using ProB.

Example. Using the previous scenario, we define a mutant:

start_BP → increase_vp → *start_BP* → manage_vp_excess

In this mutant, we try to start again the BP while the vp is increasing. This scenario is not allowed by the developed specification because the guards of the start_BP event prohibit the restarting of the BP when it is already working.

IV. Related work

The poor quality of the requirements and the exclusion of the client have a negative impact on the resulting software, see the CHAOS report[5]. More than half of the projects cost twice as much as the estimated initial budget; other projects were abandoned before being completed. The most common errors occur in the upstream phases of the project. In our approach, we interact with the client via questions all along the development.

Systems become increasingly hybrid [28]. The requirements documents used in this industry are written using informal natural language and are thus leading to requirements ambiguity [2]. Su et al. recommend to re-write these documents into two different texts : explicative and referential [27]. The first one is for understanding the problem, the second one is composed of requirements and definitions described by means of short sentences written in natural language. Even though, these short sentences add to their clarity, but natural languages are susceptible to multiple interpretations based on different writing styles. Multiple controlled natural languages help to reduce this requirements ambiguity by proposing some patterns of requirements [19]. The requirements expressed in this fashion may be documented in a client's requirements document. We use the ProR tool [14] for managing this document and propose an evolution of the requirements structure done simultaneously with the preparation for the validation activity [22].

The KAOS goal-oriented approach [30] provides a smooth transition from the informal requirements gathering phase through to the formal specification of systems components. The main idea concerns the identification of the system goals, their gradual refinement until obtaining constraints and the associated method proposed to derive the requirements in terms of objects and actions. The gap between textual requirements and formal models is explored [21], showing how Model-driven engineering can be extended to the requirements level with KAOS. The central idea is to map Goal-Oriented agents to a hierarchy of Event-B machines [20]. A design process is proposed to decompose a system level agent/machine into finer grained agent/machines based on their ability to control specific piece of information. The approach is semi-automated and tool supported by an Eclipse plug-in connecting the KAOS/Objectiver tool and the Event-B/Rodin tools. For [10], the requirements specified with SysML/KAOS goal diagrams are automatically translated into B System specifications. The architecture of the formal specification and the domain properties are automatic translated. The structural part of the formal specification is completed. The body of events is manual. We use tools such as ProR, ProB and provers of Rodin at any moment of the development and different tools helping a multi-vision of the different components.

In order to bridge the gap between requirements and Event-B specification, the authors [5] use semi-formal structures, conserving a trace between the two documents. They (i) classify the requirements according to Event-B structures, (ii) use UML-B [26] and Event Refinement Structures [6] to provide a graphical view of classified requirements and to develop a refinement structure of the future Event-B models and (iii) generate formal Event-B specifications using tools. Our approach does not depend on the used formal language and is not just for specifications: it focus on the evolution of the requirements, their corresponding specification and the links between them. This evolution is performed by means of development choices.

The approach described in [13] proposes a methodological guide for requirements and specification elicitation. This approach does not introduce new languages or formalisms. The process of requirements elicitation is independent of the used specification language. A traceability link between the requirements and the specification is maintained via *agendas* that express the summary of the method. The agendas describe and guide the development process, ensuring a certain guarantee of the quality of the product obtained through the validation conditions associated with the various steps. Our approach is a recent evolution associated with the use of the Rodin platform and the ProR tool with a semi-automation of the development.

V. Conclusion and future work

The analysis of the requirements is based on understanding and extracting the information needed for the development. Our approach allows to show the discoveries related to the problem at hand. These discoveries are mainly related to oversights, ambiguities and imprecisions. It is based on the refinement technique including the gluing notion, defining a link between concrete and abstract states by relating abstract

[5]The Standish Group Report, https://www.projectsmart.co.uk/, version published in 2014 by Project Smart.

variables to concrete ones. We have define different patterns development to write down predefined forms for the requirements which are reused in different situations relatively to its parameters. These patterns are implemented in Event-B language and TLA^+. They are for all proved complete and correct.

The validation of the specification according to the requirements document is a rigorous process treated from the analysis of the requirements and takes into account the evolution of the development. This process covers the expected behavior as well as the facts, actors, functionalities and obligations of the future system. The requirements are usually insufficient to validate the specification.

The available tools in the Rodin platform have an important role throughout the development process. These tools automate tasks like evolving the requirements, updating the glossary and refining the formal specification; it is always available. We used the ProR tool for writing the requirements, organizing them in a hierarchical structure and updating them. The glossary is automatically updated with the evolution of the specification. We use the verification and validation tools namely the proof obligation generators, the provers, the animator and model-checker ProB to ensure the correction of the specification all along the development. We use graphical tools like Event-B state machine and Project Diagram for presenting machines and contexts relations.

The limitations of our approach concern the extraction of validation elements and is based on the understanding of the requirements. It is hard to extract such elements when requirements are ambiguous. The big-sized requirements document makes fastidious this extracting task. The construction and use of the state machine diagrams should be fulfilled carefully, because this diagram extracted from the specification can be erroneous. Non-Functional Requirements have to be taken into account.

REFERENCES

[1] Jean-Raymond Abrial. B : passé, présent, futur. *Technique et Science Informatiques*, 22(1):89–118, 2003.

[2] Jean-Raymond Abrial. Formal Methods in Industry : Achievements, Problems, Future. In *28th International Conference on Software Engineering, Shanghai, China*, pages 761–768, 2006.

[3] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[4] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[5] Eman Alkhammash, Michael J. Butler, Asieh Salehi Fathabadi, and Corina Cîrstea. Building Traceable Event-B Models from Requirements. *Science of Computer Programming (Special Issue on Automated Verification of Critical Systems, AVoCS 2013)*, 111, Part 2:318–338, 2015.

[6] Michael J. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods, 7th International Conference, IFM, Düsseldorf, Germany*, pages 20–38, 2009.

[7] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. SMT Solvers for Rodin. In *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ Pisa, Italy*, pages 194–207, 2012.

[8] Edsger Wybe Dijkstra. Notes On Structured Programming. page 88, avril 1970.

[9] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT, Santa Margherita Ligure, Italy*, pages 502–518, 2003.

[10] Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau, and Amel Mammar. Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach. *Int. J. Softw. Tools Technol. Transf.*, 22(3):349–363, 2020.

[11] Martin Glinz. A Glossary of Requirements Engineering Terminology. Technical report, Standard Glossary for the Certified Professional for Requirements Engineering, Version 1.6, 2014.

[12] Fahad R. Golra, Fabien Dagnat, Jeanine Souquières, Imen Sayar, and Sylvain Guerin. Bridging the Gap between Informal Requirements and Formal Specifications Using Model Federation. In *SEFM 2018 International Conference on Software Engineering and Formal Methods, Toulouse, France*, pages 44–69, 2018.

[13] Maritta Heisel and Jeanine Souquières. A Method for Requirements Elicitation and Formal Specification. In *18th International Conference on Conceptual Modeling, France, 1999*, number 1728 in LNCS Springer-Verlag, pages 309–324, 1999.

[14] Michael Jastram. ProR, an Open Source Platform for Requirements Engineering based RIF. *In Systems Engineering Infrastructure Conference, SEISCONF*, 2010.

[15] David Chenho Kung and Hong Zhu. Software verification and validation. In *Wiley Encyclopedia of Computer Science and Engineering*. 2008.

[16] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[17] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In *International Symposium of Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[18] Atif Mashkoor. The hemodialysis machine case study. In *5th International Conference ABZ: Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 329–343, 2016.

[19] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy Approach to Requirements Syntax. In *17th IEEE International Requirements Engineering Conference*, pages 317–322. IEEE, 2009.

[20] Christophe Ponsard and Xavier Devroey. Generating High-Level Event-B System Models from KAOS Requirements Models. In *INFORSID*, pages 317–332, 2011.

[21] Christophe Ponsard and Emmanuel Dieul. From Requirements Models to Formal Specifications in B. In *ReMo2V*, 2006.

[22] Imen Sayar and Jeanine Souquières. La validation dans les premières étapes du processus de dévelopement. *Revue ISI-DAT, numéro spécial « Décisions, argumentation et tracabilité dans l'Ingénierie des Systèmes d'Information»*, 22(4):11–41, 2017.

[23] Imen Sayar and Jeanine Souquières. Bridging the Gap Between Requirements Document and Formal Specifications using Development Patterns. In *IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 116–122, 2019.

[24] Renato Silva and Michael Butler. Shared Event Composition/Decomposition in Event-B. In *9th international symposium, FMCO 2010, Graz, Austria*, pages 122–141, 2010.

[25] Renato Silva, Carine Pascal, Thai Son Hoang, and Michael Butler. Decomposition tool for event-B. *Software: Practice and Experience*, 41(2):199–208, 2011.

[26] Colin F. Snook and Michael J. Butler. UML-B: Formal Modeling and Design Aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122, 2006.

[27] Wen Su, Jean-Raymond Abrial, Runlei Huang, and Huibiao Zhu. From Requirements to Development : Methodology and Example. In *13th International Conference on Formal Engineering Methods, Durham, UK*, pages 437–455, 2011.

[28] Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Formalizing Hybrid Systems with Event-B and the Rodin Platform. *Science of Computer Programming*, 94:164–202, 2014.

[29] Axel van Lamsweerde. Formal Specification: a Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159. ACM, 2000.

[30] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[31] Hui Yang, Anne De Roeck, Vincenzo Gervasi, Alistair Willis, and Bashar Nuseibeh. Analysing anaphoric ambiguity in natural language requirements. *Requirements engineering*, 16(3):163, 2011.