The Role of Validation in Refinement-Based Formal Software Development

Jean-Pierre Jacquot, Atif Mashkoor

The Role of Validation in Refinement-Based Formal Software Development*

Jean-Pierre Jacquot[1] and Atif Mashkoor[2]

[1] Université de Lorraine & LORIA, Vandœuvre-lès-Nancy, France
firstname.lastname@loria.fr
[2] Software Competence Center Hagenberg GmbH, Hagenberg, Austria
firstname.lastname@scch.at

Abstract. In this chapter, we consider the issue of validation in the context of formal software development. Although validation is a standard practice in all industrial software development processes, this activity is somehow less well addressed within formal methods. As the needs for formal languages, tools and environments are increasing in producing real-life software, the validation issue must be addressed. In this chapter, we discuss what the place of validation within formal methods, what specific issues there are associated with formal methods as far as validation is concerned, and what tools can be used in this regard. We then present a few examples of the usefulness of validation from the case studies we have developed. The chapter is concluded with a few open research problems associated with validation and future work.

1 Introduction

This chapter discusses the role of *validation* in the context of formal software development. We borrow the definition of validation from the US Food and Drug Administration (FDA) principles of software validation[3] which state that "validation is a confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled." In this chapter, we focus on model-oriented and state-based (refinement-based) methods such as B [1], Event-B [3] or Z [25]. Such methods are based on two principles:

1. the specification of a system or software can be cast as a mathematical model expressing invariant properties on a state, and

* The work of Atif Mashkoor is supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

[3] https://www.fda.gov/downloads/MedicalDevices/.../ucm085371.pdf

2. a correct implementation can be derived by a chain of refinements of the formal specification adhering to a strict set of correctness rules.

It thus follows that, by using such a method, developers are guaranteed to produce software which is provably consistent with the specification. They have "built the software right." However, nothing guarantees that they have "built the right software." So, the verification part of the engineering effort is covered, but not the validation part. We believe that software developers will be more likely to use formal methods if they take into account this latter part.

The issue of validation within formal methods has come up only recently for two main reasons, one good and one bad. The good reason is that the research community had to focus first on hard theoretical problems, e.g., how to automate most of the verification process. Without a high level of automation, formal methods are not usable in practice. Now, the power of theories and tools allows us to tackle real-life problems, see for example [7–10]. Provers know how to discharge complex formulas, model-checkers can deal with huge spaces, and formalisms allow developers to simply express complex properties. While there is still room for much improvement, it seems safe to say that we have passed the threshold where the methods could reasonably be deployed. The bad reason is the belief that validation is not necessary: we "just" need to get all the right properties; i.e., we just need to focus on getting a complete specification from the requirements analysis. We should also mention a probable last reason: validation requires human judgment, and so is outside the formal world.

While requirements engineering is indeed a key element of any successful development, it is unrealistic to expect this activity to produce a complete set of expectations, constraints, and assumptions before the development begins: requirements evolve, decisions during development must be made, assumptions are overlooked, and not all requirements can be expressed in a specific formal method.

Our position is that validation must be part of all stages of a development conducted with refinement-based formal methods: from the initial specification, to the implementation, passing through each strategic refinement [17, 19].

The chapter is organized as follows. We first discuss what validation is and why we must use it in the context of formal software development. Then, we present the important issues associated with validation. Then, we discuss the kind of tools, existing and to be invented, necessary for validation. Last, we give some examples where validation uncovered critical problems in case studies which were already verified.

2 The Place of Validation in Formal Methods

The term "validation" is understood, slightly, differently in science and in engineering. In the former, validation consists of checking that the predictions

of a model conform with reality. In a sense, validation measures the explicative power of a model. In the latter, validation consists of checking that the produced artifact is consistent with its users' expectations. As formal methods are an engineering technique based on mathematical modeling, "validation" concerns both meanings.

In this section, we analyze validation in relation to different phases of a software development project. We discuss what can be validated and why it is important to do it.

2.1 Initial model

The starting point of any development using a formal method is a formal specification of software to produce. It consists of a set of mathematical and logical formulas. We are then in the classic scientific validation issue:

- do the formulas express properties we want to account for?
- do the formulas lead to the correct predictions?
- do the formulas cover all the properties we want to account for?

A major difference between models in software engineering and models in other engineering is that in the latter models are often more "numerical" and in the former models are more "logical." Numerical models are, in principle, easier to validate since computing with them produces numbers which can be checked against existing data or measured on physical objects. Logical models are, in principle, more complicated to validate since we need to draw all possible inferences from the formulas.

For models with few formulas, one can expect domain specialists to assess the validity of the specification through thorough reading. For larger models, the task requires the use of tools to explore the possible inferences.

2.2 During development

In refinement-based methods, a development step is a refinement: an evolution of the model which must meet very strong consistency relations between before and after the step. Those constraints have the form of theorems, or proof obligations, which must be discharged. There is then a solid safety net for the model's evolution. However, validation is necessary for several reasons: extension of the model, fixing-up of formulas, developer choices, emerging behaviors, and assessing progress.

Extension of the model Some methods, such as Event-B, allow developers to introduce new properties during refinement. On the plus side, this allows to build incrementally the model by introducing gradually its different features. The notion of observation level[4] is an attempt to organize rigorously such

[4] An observation level is a focus on a specific part of a formal model for a fine-grained analysis [15, 16].

introduction. The new formulas must of course be validated in the same spirit as for the initial model, but we must also check that they interact well with the part of the model that has already been specified.

Fixing-up formulas The general direction of refinement is to go from non-deterministic to more deterministic behaviors, and from abstract data models to implementable data structures. The proof obligations guarantee that the refined formulas are consistent, but not that they are a valid model. For instance, in Event-B, the guard of a refined event must be stronger than the guard of the corresponding abstract event. That can be cast as a proof obligation which must be discharged. But, upon success, we must also check that the new guard is not "too strong," thus discarding wanted behaviors from the model.

Developer choices Refinements, most often, require the developer to make some choices. Regularly, even good and precise specifications do not offer any clue to guide the choices. The problem is that all choices are not equal! For instance, the complete specification of the withdrawal function of an ATM is an attainable goal. There are three inputs (card, PIN-code and amount), three outputs (card, cash, account modification) and a limited set of conditions (even integrating hardware problems). But, at some time during the development, the two operations of giving back the card and the cash must be ordered. Interestingly, the first generation of ATM gave first the cash, then the card. Now, the order is reversed: one must retrieve the card to get the cash. The most probable explanation for this switch of behavior is that humans are forgetful and left many cards behind. This is the kind of assumption that is likely to be omitted in a formal model. The point is not that the problem would have been caught during the development, but that, without validation, it is nearly impossible to spot.

Emerging behaviors As refinements proceed, the model is growing in size and details. It contains more and more formulas which may interact in unexpected ways. This growth has the potential to introduce emerging properties in the model, some maybe good, some maybe neutral, and some maybe bad. Of course, it is better to avoid the bad ones as soon as possible.

Assessing development progress A (probably undesirable) feature of formal methods is their opacity: potential users and stakeholders do not possess, generally, the mathematical background necessary to read and understand thoroughly the models. This can lead to uncomfortable situations because stakeholders cannot be included in the development process [11]. This has several consequences:

– problems with requirements and unwanted behaviors are not caught before the development is finished,
– nobody, except the developers, can assess whether the development is going in a good direction,
– without continuous monitoring of the development progress, it will be more difficult to convince managers of the benefits of formal methods.

2.3 Implementation

Using formal methods does not forgo the necessity to test the final product. However, the kind of tests that are required later on are like acceptance tests where it is determined that the product meets users needs and unlike other tests which (try to) assess that the invariant properties are preserved.

Nonfunctional properties Nonfunctional properties, such as response time or maintainability, are equally important requirements. Even when they can be formalized, such as a bound on response time, they are often outside the scope of formal methods. Meeting nonfunctional properties is also often dependent on the hardware and the environment where the implementation runs, which may not be exactly known or modeled during development.

Usability For systems which interact with humans, e.g., ATMs, it may be impossible to predict all the difficulties that may arise. This should be the true role of beta-version software: validating that the system improves the users' ability to fulfill their task.

2.4 Non modeled properties

Irrespective of the considered formal method, there are some properties that are outside its expressive power. For instance, it is very difficult to model general time, causality, or reachability properties in Event-B. So, we can expect, for any real-life software, to have requirements which cannot be formalized. It is then important to test for those requirements. The tools and techniques to test are the same as those used to validate the models through execution. So, these tests can be included in the validation procedure.

As soon as a model takes care of requirements outside the expressive power of a particular method, specific tests must be setup. Further down the development, such tests must be run at every step in the spirit of non-regression testing.

3 The Issues with Validation

The validation of models is not a new idea. It is indeed an important part of traditional software development processes. However, formal methods, until recently, mostly ignored validation. We think the context of formal methods induces some specific issues about the validation process. This section discusses those.

3.1 Requirements

Whatever development method is used, the validation of a piece of software is only worth the quality of the requirements for that element of a system. Precise requirements are the goalposts against which the final product must be measured. Hence it is absolutely important to have a detailed, well written, complete, and unambiguous statement of requirements to start with.

In the context of formal methods, there are two issues with requirements. The first issue lies with the formalization of requirements themselves. A critical question is the coverage of requirements: does the final formal model takes all of them into account? To answer such a question, we need to be able to trace requirements back and forth to the formal model. There is then a need to have a good level of compatibility between the modeling languages used for requirements and for models. This is, for instance, what the ProR tool[5] proposes for Event-B.

The second issue lies with the evolution of requirements. We are not convinced that a better requirement elicitation process would make this issue disappear. Of course, better requirements must always be sought out, but they will still evolve for two main reasons: implicit assumptions and environment modifications. Software which interacts with open environments, notably humans, must rely on assumptions about this environment (ranges of value, behaviors, etc.), many of which are not formalized. Yet, once in a while, we discover that the formal models allow behaviors that are undesired. Also, the notion of environment of a system is a point of view of the system. But there are other points of view, from which the new system is a modification of the environment. By its mere existence, the new system may shift the expectations for its users or create new expectations.

The problem is actually an open research question: how to manage modifications in a formal model? Until now, there is no other option than starting again the development.

3.2 Human judgment

Validation implies that someone makes a judgment on the model. This means that users must assess whether the model meets requirements. There are two

[5] http://www.eclipse.org/rmf/pror/

ways to make such an assessment: one is to "understand" the model enough to explain its relation to requirements, the other is to run the model and analyze its output.

The former technique is adequate for models which consist of few formulas (invariant properties, events, automatons, etc.). As the size of the model grows, the interaction between the logical formulas becomes more and more difficult to analyze. For instance, we have often observed that we tend to write stronger than necessary guards for events, therefore forbidding otherwise admissible situations. Such errors are difficult to catch through reading only.

Larger models, in terms of the number of elements and formulas, are better validated by analyzing the space of legal states they specify. The analysis revolves around two kinds of questions: Which states are reachable? Are all the ways to reach a state admissible to the user or the environment? The first question concerns the coverage of requirements by the model; the second question concerns the problem of inadequate behaviors. This can be achieved by techniques akin to those used in software testing. So, we can adapt existing techniques to reduce safely the amount of executions to explore the state space.

3.3 Abstraction and nondeterminism

In using a test-based approach to validation, we are confronted with a major difficulty: how to execute the model? While it is always possible to run a state-based model by computing manually a few traces, a true validation requires to compute many more. We need an automated execution mechanism.

An executable program is a formal model. However, it is way too detailed to allow for formal verifications in general. There lies the reason which motivated the introduction of formal methods. Abstract formal models are allowed to use data which may not be implemented as such, e.g., operations with non-constructive definitions, and nondeterministic execution models, so proofs can be conducted.

Execution tools must provide solutions for two different problems. The first is a way to implement data. Since efficiency can be put aside (up to a reasonable degree), we can use canonical definitions and libraries. The second is the exploration of the state space. Both nonconstructive and nondeterministic definitions increase the size of the state space to explore. There is a threshold below which automated exploration is possible. Over this limit, we must resort to human intervention to guide the exploration.

When using an execution tool, we must be confident that the results and behaviors we observe are indeed specified by the model. The tool must be correct. Fully automated tools can be trusted more easily than tools where user's intervention is required. In both cases, we need some insurance that tools correctly implement the operational semantics of the formalism. In the last case, we also need the insurance that elements provided by users are consistent with the model. We have addressed this issue with the notion of simulation "fidelity"

which is looser than the strict behavioral equivalence relation, but ensures that observations made on the executable model can be trusted as observations on the original one [19].

4 The Tools

The actual use of formal methods is highly dependent on sophisticated tools. Formal models are too complex to be manipulated "by hand," and the operations on them are highly complex, relying on elaborate theories and extensive exploration. Verification, either through theorem proving or model checking, is not possible without tools; validation is not different. We present here a typology of tools we have found useful.

4.1 Translators

A development using formal methods and refinement will eventually produce a program written in a programming language. So, at some point, the mathematical and logical model must be translated. Then, standard testing techniques can be used to validate the final program. Sometimes, when the model is deterministic and the data has some canonical implementation, translators can be used during the development. Of course, this will generally happen when the development is near its end.

The main advantage of using a translator in a validation procedure is that the executable model is very close to the final program. Assuming the translator is proven (or at least verified), the observations made while executing the program correspond exactly to the model. Execution can be trusted without further consideration. The environment in which the model is executed is close to the environment for the final product. So, the kind of non-functional requirements that depend on the actual environment can begin to be assessed.

Of course, the main limitation is that the model must be close to implementation. This limits the use of translators to the very last stages of the development. We should strive to have the major problems with requirements been discovered and overcome before those stages.

4.2 Model checkers

The validation of a software model is to explore the state space defined by the model. Model checkers are built for that exact purpose and, indeed, can be used during the development. ProB [12], a model checker for B adapted to Event-B, has been extended by an interface control and shows the execution of the model.

The main advantage of model checkers is their solid formal foundation. When they provide an answer, we are assured that it is a true consequence of

the model. Also, their exhaustive exploration of the state space insures a full coverage of the model's behaviors.

The main limitation of model checkers is the combinatorial explosion of the state space exploration. Nondeterminism fuels the explosion, however, even deterministic systems can lead to models with a state space too large for practical application of model checking. For instance, in the landing gear case study proposed for ABZ'2014 [6], most published studies acknowledge that the use of model checkers becomes impossible when a certain level of details has been reached. Yet, the system is purely deterministic.

4.3 Animators

Animators are tools which implement the operational semantics of the formalism and are able to interpret directly the formal code. The nondeterministic features are treated by enumerating all the possible local values and picking randomly the actions to do when necessary. The enumeration can use either smart constraint solving techniques (ProB for instance), or brute force enumeration (AnimB[6] for instance).

The main advantage of animators is their close proximity to the formal model. They interpret directly the code over a "virtual machine" which is relatively easy to verify.

Animators have the same limitations as model checkers. Since they need to enumerate values, those need to belong to relatively small sets. It is possible to develop heuristics to transform nonanimatable models into animatable ones, e.g., as proposed in [14,18]. It can be done safely but at some cost: transformations generate proof obligations which must be proven, and the resulting model is only a sub-model of the original model. Some behaviors and data state may be lost, but no spurious ones are introduced.

4.4 Simulators

To make validation accessible at all times during a development, there is a need to execute very abstract and nondeterministic models on which all the preceding tools fail. The idea then is to rely on automatic interpretation on most of the model and ask the user to provide explicit definitions for nonconstructive constructs and choice restriction for the nondeterministic constructs. The rationale is that developers must begin with very abstract models, but they know in which direction the development will proceed to become implementable. Of course, the hand-coded parts provided by the users constraint and restrict the space that can be explored, but we can expect that the cut-out parts of the state space are actually irrelevant for the final software. For instance, when modeling some movements in a 3D space, we may need some distance function

[6] http://www.animb.org

which is not explicitly modeled; for practical validation, "implementing" it as the standard Cartesian distance is effective.

The main advantage of the simulation technique [26] is that models at all levels of abstraction can be validated, at least partially.

There are two limitations to the technique:

- the explored state is smaller than the specified space; we must trust users to provide solutions that explore the space which is important in practice,
- the user's additions must be consistent with the model. This is a complex issue. For Event-B, for instance, we have developed a notion of "fidelity" which captures this property [28]. It entails the generation of proof obligations which must be discharged to guarantee the consistency of the simulation.

4.5 Scenario managers

To check whether a given requirement is covered by a system, we need to set up a test situation: getting to an initial state value, firing a predefined sequence of commands or actions, and comparing the expectations against actual results and observations. To validate a model, we need to check all requirements the model is supposed to cover. So, we must have some tools to manage a large collection of scenarios.

Furthermore, for validation to be a continuous process accompanying refinements, we must check that no requirement is dropped off during a refinement. This is the standard issue of non-regression testing that faces the maintenance process and the incremental development methods.

5 Case Studies

In this section, we present a few examples of our developments where the execution of models helped us spot and eventually fix anomalies in our models. All the following examples are taken from our work with Event-B. So before discussing examples, we first introduce this formal method.

5.1 The Event-B method

Event-B is a state-based formal method for systems and software engineering. A model in Event-B is composed of three elements:

1. a state, which is a function mapping names to values. The values are inductively defined as atoms (integers, symbols, booleans), sets of atoms, and set-theoretic constructions of values. There are special notations for typical set constructions such as relations, power sets, total relations, or bijective functions. Syntactically, the state distinguishes between constants and variables.

2. an invariant, which is a logical first-order formula on the state that specifies the legal values of the state of a system. Syntactically, the invariant is a conjunction of smaller formulas.
3. a set of events, which are guarded substitutions on the state. The guard is a conjunction of smaller first-order formulas on variables of the model.

The formal semantics of the model is defined by four properties:

1. all expressions must be well-typed; the typing relation is essentially the "belongs to" set-theoretic relation,
2. there must exist at least a value of the state which satisfies the invariant. Syntactically, there is a special event (*INITIALISATION*) which sets the initial legal state,
3. any event whose guard holds can be fired (the choice is nondeterministic), each event fired from a legal state leads to another legal state: i.e., they maintain the invariant,
4. all events must be feasible, i.e., if the guard is true, there must exist a solution for all substitutions which leads to a legal state.

The model development scheme associated with Event-B is based on formal refinement. Refinement is a relation between models. A model MR is a refinement of MA if: (1) there is a gluing invariant between the state of MR and the state of MA, this gluing invariant is part of the invariant of MR, and (2) all events in MR have their counterparts in MA and have a stronger guard ensuring that a refined event can only be fired if the abstract event can.

In practice, refinements can be classified into four general kinds:

State restriction The invariant or guards of events are strengthened so the legal state space becomes smaller.

Data refinement Some variables are "replaced" by others closer to an implementable data structure.

Property introduction New variables and related invariants are introduced to model a property.

Behavior refinement New events are introduced to model new behaviors. Technically, new events must be formal refinements of the *SKIP* (do-nothing) event.

The syntax and semantics are designed so that, for each model, a set of independent proof obligations are generated. The model is verified when all proof obligations are discharged. The major advantage of refinement-based methods is that the proof of correctness of the implementation is broken into many small and relatively simple proofs which are spread out all over the development process.

5.2 The platooning case study

The platooning case study [27] was intended to assess the possibility of using Event-B to prove an existing algorithm of platooning by redeveloping it. The algorithm controls the movement of autonomous vehicles forming a platoon by using only perception of the preceding vehicle in the formation. The safety property of interest is non-collision of vehicles participating in the platoon. The development was conducted in two phases: the first phase considered only a 1-D version of the algorithm which only controls speed, the second phase considered a 2-D version which adds direction control. The examples below concern only the 1-D version.

The validation was done by using the animator Brama [23] to which a graphical interface was added. It was performed at the end of the development, on a model which included an event to set the target speed of the platoon which was not part of the specification. This unguarded event does not modify the state space or the essential behaviors of the model.

The animation uncovered two interesting problems with the model.

Oscillations During the animation, it appeared that when the platoon reached its target speed, although the leading vehicle had a smooth continuous speed, the following vehicles were alternating braking and accelerating with each cycle. Furthermore, acceleration/deceleration values increased with the position in the platoon. This is of course an unintended behavior of the system. This situation comes from the incompleteness of requirements which must include a statement about oscillation. It should be noted that the definition of oscillation in Event-B may be tricky.

Blockages When running a scenario where the platoon comes to a stop, we observed that it could not start again. Analysis of the problem reveals that the platoon algorithm was actually caught in a deadlock. The reason was that Event-B supports only integers while the algorithm uses real numbers. The slight difference in values prevented some guards to be true. The solution was to modify slightly the model to account for the numerical discrepancy. Event-B has no provision to guarantee that models do not deadlock. It is possible to define invariants which express the deadlock freeness property, but they must be introduced explicitly.

5.3 The landing gear case study

The landing gear case study [5] was introduced as part of the ABZ conference to allow comparison of formal systems and development approaches. The task is to design the control system of the landing gears of an airplane. The requirements are given in a natural language document which describes the physical components, the sequence of movements, and the timing constraints of a live system.

A very important property of the control system is to allow maneuver reversibility at all time, i.e., the pilot must have the possibility to stop and reverse the maneuver at any time. Such a property cannot be specified in Event-B. However, as the system can be described as a finite automaton, it is possible to design an exhaustive set of scenarios to check the reversibility.

All solutions of the case study based on Event-B observed the necessity to use a form of execution to check the property; all developers admitted to have reworked their models a few times to get it right.

An interesting observation about the tools is that, even if the system is finite and deterministic, automatic tools, such as ProB, stop to work as the description of the hardware elements gets more detailed.

We used JeB [19]. An important part of the code we had to write consisted in the executable models of the hardware, the sensors in particular.

5.4 The transport domain model case study

The transport model [13] is an experiment to assess the potential expressiveness of Event-B to specify domains. The requirements include the following:

- vehicles move along a network akin to a road system,
- vehicles must not collide,
- two kinds of collisions are considered: rear-endings along a path and intersection crashes,
- vehicles obey the usual kinematics law,
- travel time must be modeled,
- energy consumption must be modeled.

A very important feature of the development is the gradual introduction of the properties.

We used two kind of tools to execute the model: Brama (an animator) and JeB (a simulator). While the use of Brama prompted the design of transformation heuristics (please see [14] for more details) to make models amenable to animation, JeB allowed us to execute all models in the 13 refinement steps. Execution of the model helped us on several grounds.

Context validation Although execution tools are designed to check the behavior of the model, the technology can be used to validate that the static part (the contexts in Event-B) is a valid model of the environmental data of reality. Modeling a small village showed that we need to introduce new kinds of nodes in the network to model the entry and exit points as (infinite) sources and (bottomless) sinks.

Deadlocks identification A difficulty with our model is that it must account for gridlocks, i.e., places in the network where no movement is possible, but must also avoid deadlocks, i.e., situations where a real vehicle could move but the model prevents it. Traditional techniques for deadlocks checking do not work well as the model allows for all vehicles to be parked.

Guard debugging Most of the anomalies seen during the executions were traced back to incorrect guards in events. Most often, the guards were too strong for one of two reasons. The first is "logically" too strong, i.e., some cases were forgotten when writing the guard. The second is "numerically" too strong. As for the platooning case study, the kinematic functions are modeled with integers. So, the formulas must take into account some kind of "imprecision" due to the differences in arithmetics.

Termination checking An important implicit property is that unobstructed vehicles must progress toward their destination. Such a property is difficult to model in Event-B. Execution allows us to assess the progress.

6 Research Issues

In this section, we discuss more precisely two issues which have recently emerged as new research topics.

6.1 Interaction between models

Formal methods are very relevant for software pieces which are part of wider systems where they control hardware elements. Cyber-Physical Systems (CPS) are of this kind, and they are becoming ubiquitous in our everyday environment. The validation of software models then also requires hardware elements to be modeled and participate to the execution strategy. Except for the rare hardware pieces which act as digital state machines, most are analog devices obeying some physical laws. Thus, they are more likely to be modeled with techniques such as differential equations or frameworks like Scilab[7]. We know how to produce executable models of such artifacts, but the interaction between the executable models of software and hardware is an open problem.

The following issues must be addressed:

1. Connecting the executable models. This is a standard issue in software engineering when we need to build systems composed of units running in different environments. In the context of Event-B, several directions are investigated. The European FP7 ADVANCE project[8] explored platforms build on Functional Mock-up Interface (FMI) [21], Rodin [4], and ProB to test CPS system with multi-simulation. Generic platforms, such as MECSYCO [24], provide a basis for connecting JeB generated executable models with models developed in other languages. Preliminary studies we have conducted ahve shown the potential of such approaches.

[7] http://www.scilab.org
[8] http://www.advance-ict.eu

2. Equalizing the abstractions. At a given refinement step, the formal model is an abstraction of actual software. The model considers the outside environment from this abstract point of view and must sees it at the same abstraction level. We then need to bring the hardware simulation models up to the same abstraction level. There are two basic possibilities. Either a model of the hardware at the appropriate level is built by the specialists of these artifacts, or, an "abstracting interface" is wrapped aroung a detailed model of the hardware to make it interact at the required abstraction level.
3. Which ever of the preceding two techniques is used, there remains the essential question: can the observation on the simulation be trusted? This boils down to the fidelity of the hardware simulation model to the actual hardware piece.

6.2 Relation between refinement and validation

In formal methods, such as Event-B, the notion of refinement was introduced and formalized to support the incremental verification of the final software. While each refinement is a complete model by itself and its consistency must be proven, the syntactic and semantic structures have been defined so that only proof obligations associated with the newly introduced elements need to be discharged for the proof to be complete. The proof that the final software conforms to its specification progresses along with the proof of each refinement. The verification is then monotonic with respect to refinements. Such a monotonicity would also be desirable for validation. However, much research needs to be done to achieve this goal. Several points must be established:

1. How to identify and associate a particular refinement with the "new" requirements being taken into account?
2. How to associate requirements and test scenarios?
3. How to generate new test scenarios for the refined model from test scenarios for the abstract model? This is related to the issue of non-regression testing: we need to be able to "replay" tests to check that behaviors are valid "refinement" of behaviors validated on abstract models. A definition of scenarios which can integrate a notion of abstraction/refinement is then required.

At present, points 1 and 2 are more a, complex, engineering problem than a research issue. Tools, such as ProR, already allow the elicitation and formalization of requirements in a format compatible with Event-B models in Rodin [22].

Point 3 requires first to have a formal definition of scenarios compatible with refinement. Since the semantics we use to define fidelity, i.e., the relation between formal models and executable models, is based on traces, we can consider scenarios as traces. More precisely, we can define scenarios as constraints on traces such as succession of events (from a given refinement level), succession of states, and any mix.

7 Conclusion and Future Work

Although we did not start from this point of view, we have come to realize that our work on validation led to ideas and tools that are close to the Agile spirit [20], at least the Good ideas as identified by Meyer: short development cycles, extensive "testing," and permanent user validation.

The similarity between the short development cycles (the "runs") of Agile methods and the short refinement steps as advocated in [2] is striking. In both cases, the development proceeds through a sequence of correct models, either formal or executables. Of course, the notion of correctness is very different in each case. By including validation of the formal models, we can get even a closer analogy as users can be involved and assess the adequacy of the requirements all along the development.

Behind the practice of "permanent testing" of Agile methods is the idea that, once established, the correctness of a feature must always be guaranteed. Refinement-based formal methods ensure this property by other means, and, too often, despise tests. However, testing has other roles which are useful in formal methods. The first role concerns the verification of the properties that cannot be modeled within the method. The second role is the validation that the requirements have been met.

The possibility for users to interact with models during the whole development is essential to ensure that, at the end, software will meet the actual, useful, requirements. Requirements' evolution is a hot research topic, particularly in the context of formal methods. At present, we do not know how to refactor a formal development when a requirement changes. So it is very important to detect as early as possible a problem with requirements: either missing, incomplete, or mis-adapted.

The diffusion of Agile methods in industry is a model we should strive for formal methods. As discussed above, by introducing validation into each requirement step, we hope to get the positive arguments to counter those used to put formal methods aside.

References

1. Abrial, J.R.: The B Book. Cambridge University Press (1996)
2. Abrial, J.R.: Formal methods in industry: achievements, problems, future. In: Proceedings of the 28th international conference on Software engineering. pp. 761–768. ICSE '06, ACM, New York, NY, USA (2006)
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
4. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: 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. Boniol, F., Wiels, V.: The landing gear system case study. In: ABZ 2014: The Landing Gear Case Study, Communications in Computer and Information Science, vol. 433, pp. 1–18. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-07512-9_1
6. Boniol, F., Wiels, V., Ameur, Y.A., Schewe, K.D.: ABZ 2014: The Landing Gear Case Study Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z Toulouse, France, June 2-6, 2014, Proceedings, Communications in Computer and Information Science, vol. 433. Springer (2014)
7. Bormann, J., Lohse, J., Payer, M., Venzl, G.: Model checking in industrial hardware design. In: Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference. pp. 298–303. DAC'95, ACM, New York, NY, USA (1995), http://doi.acm.org/10.1145/217474.217545
8. Butler, R., Caldwell, J., Carreno, V., Holloway, C., Miner, P.S., Di Vito, B.: NASA Langley's research and technology-transfer program in formal methods. In: Computer Assurance, 1995. COMPASS'95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on. pp. 135–149 (Jun 1995)
9. Cimatti, A.: Industrial applications of model checking. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M. (eds.) Modeling and Verification of Parallel Processes, Lecture Notes in Computer Science, vol. 2067, pp. 153–168. Springer Berlin Heidelberg (2001), http://dx.doi.org/10.1007/3-540-45510-8_6
10. Kaufmann, M., Moore, J.: An industrial strength theorem prover for a logic based on Common Lisp. Software Engineering, IEEE Transactions on 23(4), 203–213 (Apr 1997)
11. Kossak, F., Mashkoor, A., Geist, V., Illibauer, C.: Improving the understandability of formal specifications: An experience report. In: Salinesi, C., Weerd, I. (eds.) Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol. 8396, pp. 184–199. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-05843-6_14
12. Leuschel, M., Butler, M.: ProB: An Automated Analysis Toolset for the B Method. Journal Software Tools for Technology Transfer 10(2), 185–203 (2008)
13. Mashkoor, A.: Formal Domain Engineering: From Specification to Validation. Ph.D. thesis, Université de Lorraine (Jul 2011), http://tel.archives-ouvertes.fr/tel-00614269/en/
14. Mashkoor, A., Jacquot, J.P.: Stepwise validation of formal specifications. In: 18th Asia-Pacific Software Engineering Conference (APSEC'11). pp. 57–64 (2011)
15. Mashkoor, A., Jacquot, J.P.: Utilizing Event-B for Domain Engineering: A Critical Analysis. Requirements Engineering 16(3), 191–207 (2011)
16. Mashkoor, A., Jacquot, J.P.: Observation-Level-Driven Formal Modeling. In: High-Assurance Systems Engineering (HASE), IEEE 16th International Symposium on. pp. 158–165 (2015)
17. Mashkoor, A., Jacquot, J.P.: Validation of formal specifications through transformation and animation. Requirements Engineering pp. 1–19 (2016), http://dx.doi.org/10.1007/s00766-016-0246-6
18. Mashkoor, A., Jacquot, J.P., Souquières, J.: Transformation Heuristics for Formal Requirements Validation by Animation. In: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert'09). York, UK (2009)

19. Mashkoor, A., Yang, F., Jacquot, J.P.: Refinement-based Validation of Event-B Specifications. Software & Systems Modeling pp. 1–20 (2016), http://dx.doi.org/10.1007/s10270-016-0514-4
20. Meyer, B.: Agile! The Good, the Hype and the Ugly. Springer Verlag (2014)
21. Savicks, V., Butler, M., Colley, J.: Co-simulating Event-B and Continuous Models via FMI. In: Proceedings of the 2014 Summer Simulation Multiconference. pp. 37:1–37:8. SummerSim'14, Society for Computer Simulation International, San Diego, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2685617.2685654
22. Sayar, I., Souquières, J.: La validation dans le processus de développement. In: Actes du XXXIVème Congrès INFORSID, Grenoble, France, May 31 - June 3, 2016. pp. 67–82 (2016), http://inforsid.fr/actes/2016/INFORSID2016_paper_3.pdf
23. Servat, T.: BRAMA: A New Graphic Animation Tool for B Models. In: B'07: Formal Specification and Development in B. pp. 274–276. Springer-Verlag (2006)
24. Vaubourg, J., Presse, Y., Camus, B., Bourjot, C., Ciarletta, L., Chevrier, V., Tavella, J.P., Morais, H.: Multi-agent Multi-Model Simulation of Smart Grids in the MS4SG Project. In: Demazeau, Y., Decker, K.S., Bajo Pérez, J., de la Prieta, F. (eds.) PAAMS'15. Lecture Notes in Computer Science, vol. 9086, p. 12. Springer, Salamanca, Spain (Jun 2015), https://hal.inria.fr/hal-01171428
25. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)
26. Yang, F., Jacquot, J.P., Souquieres, J.: JeB: Safe Simulation of Event-B Models in JavaScript. In: Software Engineering Conference (APSEC), 20th Asia-Pacific. vol. 1, pp. 571–576 (2013)
27. Yang, F., Jacquot, J.P.: Scaling up with Event-B: A case study. In: Bobaru, M., Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods, Lecture Notes in Computer Science, vol. 6617, pp. 438–452. Springer Berlin Heidelberg (2011)
28. Yang, F., Jacquot, J.P., Souquières, J.: Proving the fidelity of simulations of Event-B models. 15th IEEE International Symposium on High-Assurance Systems Engineering pp. 89–96 (2014)