



RADIO SCIENCE FOR HUMANITY

Formal methods and Abstraction for a Safe and feasible Design *Méthodes formelles et abstraction pour une conception sûre et faisable*

Sophie Coudert¹ and Dominique Legros²

¹Institut Mines-Télécom, Télécom ParisTech, France, sophie.coudert@telecom-paristech.fr

²ENGIE INEO UTS, France, dominique-legros@engie.com

Keywords: Formal Methods, Abstraction, Refinement, Design

Mots-clefs: Méthodes formelles, abstraction, raffinement, conception

Abstract:

Formal methods have developed theoretical and practical tools for software engineering to improve the safety and reliability of computer systems. Complexity of implementation remaining problematic, these developments have, among other things, focused on the structuring of systems and problems, by means of abstraction, refinement and decomposition. This knowledge on how to segregate questions in order to deal with complexity has a range which is going beyond the scope of design and software engineering. This paper gives an overview of the unified mathematical basis approach in natural language. An example of formal method applied to an operational system design illustrates these concepts. We conclude with a discussion about the methodological aspects and possible extensions of application fields of these techniques.

Résumé:

Les méthodes formelles ont développé des outils théoriques et pratiques pour le génie logiciel afin d'améliorer la sûreté et la fiabilité des systèmes informatiques. La complexité de mise en œuvre posant problème, ces développements ont entre autre porté sur la structuration des systèmes et des problèmes, au moyen de l'abstraction, du raffinement et de la décomposition. Ces connaissances sur comment séparer les questions pour aborder la complexité ont une portée qui dépasse le génie logiciel et la conception. L'article fournit une approche abstraite unifiée des bases mathématiques générale communes à ces techniques, en langage naturel. Les concepts sont illustrés par un exemple d'application d'une méthode formelle à la conception d'un système. Nous concluons par une discussion sur les aspects méthodologiques et les extensions possibles du champ d'application de ces techniques.

1 Introduction

Formal methods are often described as a set of mathematical tools and techniques for software engineering. This is restrictive but not false. Historically, they have first been proposed to increase the reliability of software development. They afford mathematical languages allowing to write precise descriptions of requirements and systems: the so-called “formal specifications”. By that means formal methods bring additional rigor in design processes. Specifications give access to a lots of tools allowing proving properties about system, to make static analysis, to animate models or prototypes, to ensure respect of specifications by software implementation, etc. In short, formal methods potentially provide, during the design process, the strength of computer monitored mathematical proof applicable to selected aspects of the system. However, the counterpart is the concomitant effort. Mathematical proof is demanding a lot and even more when computerized. Indeed, because computing is not leaded by physical laws, any basic reasoning step must be explicit and without any shortcut.

This inhering complexity cumulative with the growing complexity of systems over the last few decades has been one of the main obstacles to the use of formal methods. Because of this, many researches about formal approaches focused on solution and techniques to overcome it, mainly from “divide and conquer” principle. Thus important knowledge and expertise have been developed in the domain about related subjects like abstraction, refinement, composition and decomposition. To preserve the basic nature of the approaches, all issues have been studied from a mathematical point of view, thereby resulting solutions and responses are mathematically sound. The scope of this knowledge is beyond the mere enhancement of software engineering methodologies. It concerns the good understanding of suitable structuring for separate analysis of different aspects of systems in general. As the typical objective is to ensure properties about systems, “suitable” means that properties about components,

parts or aspects may not be insidiously lost when combining or considering the whole system. Of course these approaches do not make all difficulties disappear, but they afford tools to reduce them, to avoid mistakes and to offer well identified guarantees. Since fully proving of all aspects of a system is generally out of reach, formal proving methods are often partially used. Lots of tools have been implemented to provide automating and assistance during design process. According to techniques and problems, they can offer proving, identifying of lost and preserved properties, conditions for safe combining, etc. Formal methods also offer support for complementary validation approaches such as prototyping or fast simulation for example.

From wider perspectives formal methods are a continuation of developments around mathematical logic. There is no fundamental frontier between their theoretical basis and those of artificial intelligence, expert systems or even databases. Originally, logic is not a tool for software or computer science. Moreover, the mathematical description of languages which is the base of modern computer system developments was initially proposed by N. Chomsky, who is linguist. The alliance between formal methods and computer science mainly originates in two historical reasons. First these approaches have widely been initiated by mathematicians coming to computer science. Second, the interest of using them strongly relies on IT support. Thus they have been developed in this field, progressively moving from software reliability to hardware/software system design, with the associated structuring aspects, methodologies and dedicated tool-kits. Progressively, their scope widens, for example, recent trends around cyberphysical systems ([1]).

The results obtained from four decade of formal methods development may interest other disciplines, other kinds of systems. Modeling and structuring do not only concern information technology. Thus this paper provides a short overview of the main fundamental basis of formal approaches associated to the issue of design. It also discuss the issue of methodology, as human judgment is essential to obtain relevant formal models and because extending the scope of formal methods to other domain supposes to afford good support to experts. The presented ideas are then illustrated on a simplified example. Section 1 presents the base of formalization and section 2 presents the structuring aspects. These two sections present the main ideas in informal way, using natural language to make content accessible to non-specialists. They are illustrated by toy abstract examples. Section 3 provides a more concrete simplified illustration and section 4 discuss methodology before concluding.

2 Formal Reasoning

Mathematical modeling opens the door to mathematical proving which is considered to be rigorous. Formal approaches go further by matching mathematics and logic, leading to mathematical logic. In the first half of the 20th century, the Bourbaki group began to develop a structured presentation of mathematics ([2]), a main contribution to “modern mathematics”. This unified view of mathematics (a very rigorous building of mathematical space) led to address the issue of mathematical foundations in terms of set theory (and later, extensions), axiomatizing and rigorous reasoning approaches in order to infer properties from axioms. It offers non-ambiguous semantics and widely shared notations, allowing reliable reasoning. This rigor was put in practice while essentially transmitted by mimicry or natural language discussion. On the other hand, logic (in Greek, *λογική*/logikê: reason/language/reasoning) was born of an early interest for identifying the right schemes of reasoning. Previously Aristotle (in *Organon*) proposed a classification of right schemes of reasoning, in opposition to fallacies or false reasoning, and in particular, sophism in politician rhetoric. Mathematical logic combines this reflexive view on human reasoning with the aim of rigor of new mathematics, resulting in mathematical characterization of sound reasoning with the successive contributions of numerous scientists merging logic, mathematics and philosophy like Boole, Frege, Russell [3], Carnap [4], Quine [5], Gödel [6], Hilbert, etc. Finally, mathematical logic led to formal logic by identifying how sound reasoning can be defined as applying rules entirely characterized by their form. Correctness of such reasoning is ensured by the proof of their soundness with respect to mathematical semantics. This reduction of proof soundness to the mere respect of forms opened the door to machine monitoring but also automation in specialized cases. So the formal methods were born. In order to enhance software reliability, computer scientists have developed many different formal techniques, dedicated languages, rule systems and sound algorithms. Authors tried to propose a general unifying view of specification formalisms for relying them, as for example [7, 8, 9], using algebraic approaches and category theory. As we can’t present all these developments and associated theories here, we provide an informal presentation of the main aspects underlying formal reasoning techniques, with some very pedagogical examples.

As illustrated by figure 1, formal approaches comprise at least a mathematical definition of a language making it possible to express properties and a mathematical definition of mathematical semantics for this language. Such semantics often takes the form of models, with a non ambiguous characterization of the verification of properties by models. A very usual notation for this is $M \models P$ meaning that property P is true in model M . An inference system allows to deduce properties from other properties. Usually $P \vdash Q$ denotes a rule allowing to deduce property Q from property P . Such a rule is often a scheme allowing multiple instantiations. For example, $n : Nat \vdash 2 * n : Even_Nat$ may be applied to 5: $5 : Nat \vdash 2 * 5 : Even_Nat$.

Before using an inference system (generally a set of rules), we must trust it. For this, we must prove that each rule is sound, i.e that for any model M satisfying $M \models P$, we have also $M \models Q$, which is called the “correctness” of the system. In other words, $models(P) \subseteq (models(Q))$ where $models(x)$ denotes the set of all models M such that $M \models x$ for any property x . With this kind of system, it is possible to chain rule applications: if $P \vdash Q$ and $Q \vdash R$, then any model of P is also a model of R . Indeed, mathematically, $models(P) \subseteq (models(Q))$ and $models(Q) \subseteq (models(R))$ implies $models(P) \subseteq (models(R))$. $\vdash^*$ often denote multiple-step deduction. Here we have $P \vdash^* R$. Correctness is essential. The reverse property is “completeness”: all what is true can be proved by the set of rules, i.e if $models(P) \subseteq (models(Q))$ then $P \vdash^* Q$. Not all systems are complete and completeness is generally difficult to prove. In particular we often renounce completeness to obtain automatic proof. Decidable systems (i.e automatable) generally address restricted sets of properties and choices are compromises.

The complete mathematization prevents mistakes. In our *Nat/Even_Nat* example, “ $\in$ ” is interpreted by the set membership “ $\in$ ” and a model consist in an interpretation of the symbol n by an element (of some set), but not any set may be used as interpretation for *Nat*. The rule holds for Natural Numbers. It also holds for Integers (positives and negatives). It does not hold for Real Numbers. Then the obligation to provide a proof of correctness ensures relevant application of the rules.

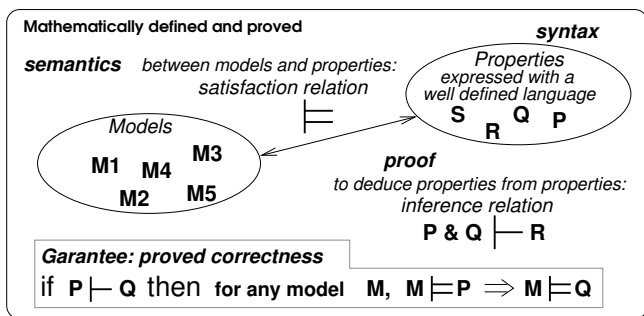


Figure 1

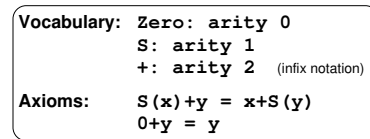


Figure 2

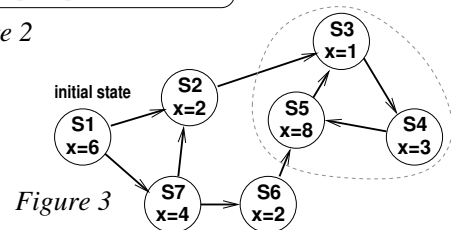


Figure 3

Axiomatic specifications ([10]) are a direct illustration of this general form of formal approaches. For example, equational specifications comprise a vocabulary (function symbols provided by an arity) and equations between terms on this vocabulary as axioms. Consider the example on figure 2. Intuitively, the unary function symbol S may be interpreted as “successor”. Applying the (sound) inference rule “ $a=b \ \& \ b=c \vdash a=c$ ” twice, $S(0)+0=S(0)$ may be inferred. Considering a mathematical definition of the sum of integer, we can prove that it respects this specification. Moreover the deduction can be automatized and thus may be used to automatically compute these sums (it is of course not efficient and success relies on a good orientation of equation, which does not exist for all problems).

Another common formal technique is model checking ([11]). Models are finite state machines which may be represented by graphs with states and transitions. Formulas are temporal properties. There are many languages to describe such automata and properties. Figure 3 is a basic illustration of such automaton. On this example may be considered as a property: it is always possible to finally go through a state where $x=2$ (in CTL logic: “ $AF(x=2)$ ”). This model doesn’t verify it because of the surrounded part of the graph on figure 3. On the other side, “ $AF(x=3)$ ” is verified. Model checking takes benefits of this particular kind of model to propose proved algorithms for verifying that a model satisfies a property. In other word, it provides proved techniques to decide the $\models$ relation. These techniques go through all states of the finite state machines. Notice that there are also correct inference relations $\vdash$ for temporal logic, allowing to deduce temporal properties from temporal properties. Such inferences address the same kind of models but offer a different service.

Although they are very simple, these examples are a good illustration of the main modeling approaches used for software specification. Static descriptions may be used for reasoning inside a state, or to describe functional tools such as a library for integer, for example. Temporal approaches allow to describe the dynamic aspects of systems, their state progression. There are logic for other paradigms but they are more rarely used for software issues and thus in the field of formal methods. They may however be relevant in future evolutions.

Automata can be directly provided by human by way of dedicated languages, to provide an abstract view of system behavior and reason about it. They also may be generated from system sources to check the real system. Indeed, most usual hardware/software system may be seen as finite state machine implementing discrete sequence of atomic calculus. In the same way, axiomatic specifications may be used to express requirement and then confront it with actual implementation. They may also be much closed to actual implementation when considering functional programming, for instance. The main problem with these approaches is complexity. The more the system is complex and its description detailed, the more the number of states of finite state machines explodes making model checking out of order. The same is true of axiomatic approaches, due to the size of the vocabulary, the number of formulas to describe relations between concepts and the number of proofs to

do. Moreover, the partial automation provided by some proof assistant often fails when considering too many properties. Thus complementary techniques are required to reduce complexity.

3 Divide and Conquer: Decomposition, Abstraction and Refinement

As complete full detailed models are usually unpracticable, partial models must be considered. Parts may be components. They represent different aspects or perspectives or levels of details. Studying parts separately allow strong simplifying. The difficulty is then to control the combination. It is customary to distinguish two main structuring orientations although it is somewhat schematic. Roughly speaking, the horizontal one addresses parts as components and the vertical one addresses parts as levels of details.

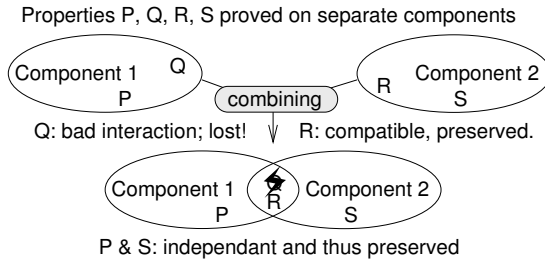


Figure 4

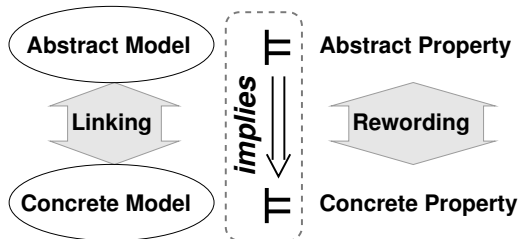


Figure 6

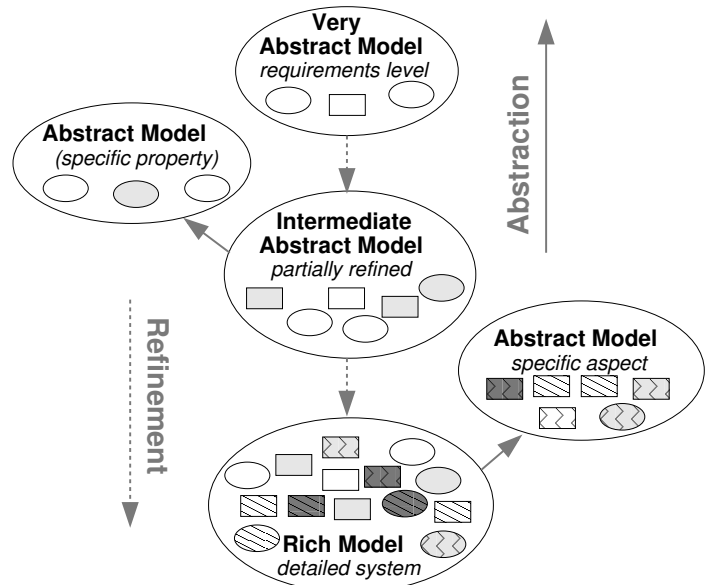


Figure 5

Horizontal structure is inherited from software engineering field, where different modules of a system may be developed separately and by different teams. Thus, each module has its own development cycle, specification and associated validation process. It is even more relevant for reuse issues, when putting together components previously developed for other projects or when managing libraries. This approach is suitable to reduce the number of human mistakes by making developers to focus on well delimited problems without considering the whole system. Unfortunately horizontal combining is often not suitable for proof issues. As schematically illustrated by the figure 4, bad interactions can emerge while combining modules. The properties which could be proven when considering components separately are lost at system level. For example, a resource can be sufficient for one component but not for two working simultaneously. Formal methods offer tools to avoid dangerous combinations. They can provide conditions on components to be compatible for combining. Such conditions are generally restrictive. Formal techniques can identify proofs to redo or complementary proofs to be provided when combining, but it often address most of the proofs. This approach remains suitable for strongly independent component or strongly local properties. In particular, this is suitable for components with no shared state, such as merely functional libraries using strictly separated work-spaces for their computations.

Vertical structure originally concerns refinement. Here refinement is considered together with its methodological counterpart: abstraction. Refinement and abstraction have similar underlying mathematical relations and main objective. The idea is to keep only in models information that is relevant with respect to addressed properties in order to simplify proof or check process. For example, characteristics of objects in a queue may be ignored wether the focus is on the size of the queue. The difference between abstraction and refinement is methodological. Abstracting consists in extracting relevant aspects from a detailed system to build a simpler one: the lower level preexists the highest one. Refining is the inverse: adding details to an abstract model thus going toward final implementation. This is illustrated by the figure 5, with shapes, gray and patterns representing aspects. Not all aspects are always fully formalized. An abstract model can be directly produced to study a property by someone having a good knowledge of the relevant part of the system but no preexisting detailed model. Abstract model can be produced early in the design process to avoid high level error before developing. The ensuing refinement process may be informal.

When using formal models or specifications at different levels of abstraction, formal methods provide guarantees about the relevance of the abstraction (or refinement). Proving properties on abstract models is only relevant if it ensures that these properties hold on concrete models. This defines the notion of correct refinement or

abstraction, schematized on the figure 6. To characterize it, we must describe the link between abstract and concrete models. If abstraction simply erases some components, it may simply be the correspondence of objects with the same names in both models. But it is sometime more complex as the representation generally change. For example, the concrete model of a house is composed of walls with doors and windows associated with a roof. Language also evolves with representation and properties may be reworded. The correctness of an abstraction with respect to an abstract property Q reformulated Q' at concrete level is that if Q holds for the abstract model, then Q' holds in the concrete one. Formal approaches provide abstraction/refinement relations with these expected characteristics and use them to enhance proving power and design processes. Model checking is strongly enhanced by abstraction which reduces the number of states of automata. Refinement approaches allow coping with system properties without considering all implementation details.

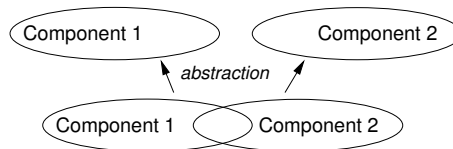


Figure 7

We have presented here fundamentals about formal proof and structure widely studied during formal methods development (see for example [11, 12, 13, 14]), that are their underlying theoretical basis. These fundamentals provide general knowledge about relevant structuring to cope with complexity while preserving properties of the parts at the system-wide level. From the general mathematical point of view of this section, distinction between horizontal and vertical structure does not really exist. Indeed, as suggested by the figure 7, components may be seen as abstractions of systems or systems may be seen as co-refinements of components. The distinction is more a methodological concern than a mathematical concern. Given that formal methods were developed as a support for enhancing system reliability, methodological aspects are also important issues. Formal approaches propose many different languages, formalisms, specialized tools relying on complementary mathematical support. For example some techniques handle timing information based on a mathematical representation of time. Formalisms provide different concrete structuring primitives dedicated to specialized aspects. For example, connecting different components through their communication channels does not pose exactly the same problems as combining different features within a same component. Like abstraction, specialization makes work simpler and leads to a better efficiency, by taking benefit of specific aspect properties without handling useless features of these. Formal tools are often provided with associated methodologies and dedicated environments.

4 Illustrated Application to Formal Design

Methodologies, in particular design methodologies, often merge formal and non-formal techniques, providing a specific support to a specific development domain. The purpose is not to present here a heterogeneous specialized methodology, but to show the contribution of formal approaches to safe design, and in particular, how abstraction or refinement are used to deal with complexity. This section addresses more deeply general principles of previous section by means of illustrations. For this we use the Event-B method ([15]) on a pedagogical example. This formal method supports a fully formalized refinement process for systems and allow to show questions and issues related to refining/abstracting and decomposing in a rather general way. The very simplified and partially developed example gives an overview of the methodological process consisting in progressing from a high level description of a problem to a low-level description of a solution to this problem.

4.1 Event-B in Brief

Event-B Machines. Event-B specifications are based on the notion of “B machines”. A “B machine” (simply called machine in the continuation of text) comprises Four main parts: 1) a vocabulary V (sets, functions, etc) with characteristics described by means of properties P , 2) a state S consisting in a set of state variables, 3) an invariant I which is a property about the state that must hold at any time and 4) a set of events $E = \{e_0, e_1, \dots, e_n\}$ which are the inputs affecting the state. Events are the only way for the state to evolve. Described systems are discrete and not continuous. One of these events, here e_0 , is the initialization which occurs once, as first event. Each event e_i has a guard g_i expressing the conditions on state in which the event can occur (except e_0 : initialization necessary occurs). Events are in general non-deterministic: they describe a set of possible state changes and an occurring event actually executes one of them. This allow to describe allowed behaviors at abstract level and refinement will progressively add implementation choices, i.e. precisions about actually executed changes.

A machine has an associated set of proof obligations, automatically generated by tools associated to the method (atelier B, Rodin). The mathematically well defined semantics of Event-B ensures that after these proofs have been done, it is sure that invariant holds in all states. Intuitively, one must prove that initialization establishes

invariant $([e_0]I$: after e_0 , I holds) and that any events e_i preserves it ($P \& I \& g_i \Rightarrow [e_i]I$: if I holds and e_i can occur, then I holds after e_i). Properties are expressed by way of a wide subset of set theory. Tools provide proof assistants helping to formal proofs. Model checking may be used when described state machines may be seen as finite for a property under study.

With respect to the concepts of the previous section, the dynamic part (S, E) can be considered as a model, the static part (I) as the properties, and the proof obligations correspond to the $\models$ relation. As the V part is static (not modified by events) no preservation property has to be proved about it but it must be carefully written because contradictions in this parts compromise guarantee (some consistency proof obligation are generated, in particular type checking, but it is partial: consistency is not decidable in general).

Event-B Refinement. Refining a machine (considered here as the abstract model) is defining a new machine (considered here as the concrete model) in the same way. To be a refinement of the abstract machine, the new machine must have some precise characteristics and properties. Variables of both machines are disjoint and the concrete invariant expresses the linking relation between abstract and concrete states. For example a force vector in an abstract variable v may be refined by two concrete vectors v_1 and v_2 with property $v = v_1 + v_2$ in concrete invariant. Each abstract event, modifying abstract state, has a corresponding concrete events modifying concrete state that must respect abstract event from two standpoints, taking linking relation into account. First any possible concrete state change must be a state change allowed by the abstract event: concrete changes are included in abstract changes. Second any possible occurrence of the concrete events must be a possible occurrence of the abstract one: concrete guards must be at least as restrictive as abstract ones. These restrictions ensure the absence of new concrete behavior on events that already exist at abstract level. Thus properties that hold in any state at abstract level also hold at concrete level without additional proofs. New events can appear in concrete machines. They only modify variables which are not linked to abstract ones: so they do not introduce unexpected changes on abstract state. These new elements may concern other parts of the described system. They also may refine abstract ones: for example by describing the steps of a calculus working on a local state, before updating the abstract state by way of a refined events (thus not new) using the result of this calculus saved in a local variable to put it in a linked variable. Some additional properties are also required for more technical requirements. For concision, they are not developed here. As for abstract machines, proof obligations are associated to refinement machines and once the proofs are done it is certain that abstract properties remains true at concrete level even after several refinement steps.

Event-B Guarantee. The mathematical foundations of the method have been (manually) proved by its creator J. Abrial ([16, 15]). Event refinements follow the principles of [13]. As the method consists in a proven process, theoretically, guarantees do not depend on specifiers skills. Only success depends on specifiers skills because finding a suitable decomposition to make proofs feasible is not trivial. The original B-method ([16]) was designed for software development, so the last refinement step produced the final algorithms, i.e. the expected system itself. The ability to obtain these final algorithms ensures that the model exists and that no consistency question remains about initial hypothesis (for example the properties about abstract vocabulary). With event-B ([15]), the relation with the real system is more flexible. Thus it may more rely on human expertise. Human must validate specification, i.e. ensure that the description is relevant with respect to real system. Human may also have to certify consistency of some remaining hypothesis. The counterpart is that the scope of the method is wider and that the method itself more powerful not in term of provability but in term of feasibility. Indeed structuring models here means in fact structuring the proof and despite the method be dedicated to system development, it may be used for many kind of proofs about systems.

4.2 Illustrating Design with Event-B

The following example illustrates the transition from high-level system requirements toward the implementation of low-level systems, i.e. the transition from the application properties to be respected toward a reliable technical solution. Due to its data acquisition capabilities, interest of drone usage is very high, to obtain rapid mapping of a particular area, for instance [17]. However, this type of machine presents also a danger for environment and people. Indeed, its weight (10-15 kg for a light drone), its power (1500 to 2000 Watts) make it a dangerous object in case of fall over urban environment. It is therefore important during the design of the drone to ensure that its behavior is that expected and that it goes into safety mode if its operation becomes potentially dangerous (for example landing emergency and switching out of service) [18]. For pedagogical reasons, the illustration used in this article is deliberately simplified. The obstacle avoiding function is only considered in the axis of displacement.

As illustrated by figure 8, this drone only moves in a straight line. Obstacles only appear ahead and don't move. Obstacles above a certain distance "MaxDist" are ignored. Only wind and motors decide how the drone moves. And a crash occur when distance between the drone and an obstacle falls below 1 (abstract integer unit).

The abstract high-level machine (figure 9) only contains the minimal data to express the "No Crash" property. A Boolean OBST indicates the presence of an obstacle closer than MaxDist, and in this case DIST is the distance

of the obstacle (else DIST doesn't matter). All moving of the drone are represented by the event `move`. At this level of abstraction, we allow any movement that does not cause a crash. Thus `move` consists in moving to any new position that respects the invariant. Indeed Invariant contains the safety property by requiring DIST to be greater than 1. In this specification, variables denote “real world” data: the real distance and the real presence of a close obstacle: we are describing requirement without any consideration about solution (such as sensors for example). As expected, the abstract specification is simple and easy to validate by an expert of the application domain, even if it is not an expert of the Event-B method. Moreover proofs are trivial: initial and new positions after `move` directly respects invariant. Two refinement steps are presented in the two following paragraphs. They are presented separately for readability but are successive in real specification, merging changes.

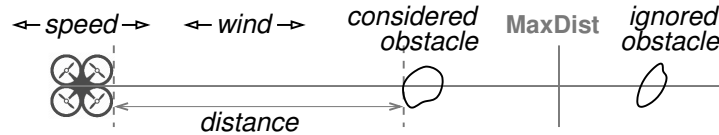


Figure 8

$Prop(x,y) = x \text{ in } \{true,false\} \text{ \& } y \text{ in } 1..MaxDist$
Machine NoCrash
Variables: DIST, OBST
Invariant: $P(DIST, OBST)$
Initialisation: DIST, OBST receive any values (O,D) such that $Prop(O,D)$
Events:
`move` == DIST, OBST receive any value (O,D) such that $Prop(O,D)$

Figure 9

$newdist(dst,spd,wnd) = (dst - (spd + wnd))$
REFINEMENT Speeds
Variables: SPEED (and WIND, not detailed)
Invariant: SPEED in $-MaxSpeed .. MaxSpeed$
 $OBST = true \Rightarrow MaxWind + DIST < SPEED$
Events:
`move` == (cases without obstacle are omitted)
for a speed S such that
 $S + MaxWind < newdist(DIST,SPEED,WIND)$,
SPEED receive S and
DIST receives $newdist(DIST,SPEED,WIND)$

Figure 10

This first refinement (figure 10) introduces the way the new position (i.e. distance to a fixed obstacle) after a move is decided. It results from ground speed which is the sum of wind speed and air speed (implicitly due to motor). So we have a new variable WIND (for wind speed) and a new variable SPEED (for air speed). Moreover we automatically have a DIST variable with implicit invariant $DIST_{abstract} = DIST_{concrete}$. It is a trivial linking relation between abstract and concrete state automatically provided by event-B. WIND changes are not detailed here but exist. Now, `move` computes the new distance considering SPEED and WIND and “chooses” a new speed (WIND represents the average wind speed since last move). In order to be safe we have a constraint on the chosen new speed avoiding hit in the worst case, when wind speed before next move is the maximal. Thus we preserve the high level safety property and have a correct refinement. Omitting this constraint makes proofs not feasible (because proof obligations are then false) and thus the methods ensures correctness. The same reason makes it necessary to consider compatible MaxWind and MaxSpeed in order to be able to stop the drone. Notice that this modeling implicitly relies on regular occurrences of `move` allowing to consider speeds as distances. Precision about time may appear in further refinement (not developed here).

This second refinement of the “No Crash” machine (figure 11) introduces technical variables in state. To detect obstacles and their distance the drone uses sensors. Variable `obst` indicate if an obstacle has been detected and in this case, variable `dist` contains the detected distance. Now the `move` event updates both tuples of variables in parallel. As for the initial specification, the invariant is trivially respected as the specification is the minimal one: any change that respects invariant is allowed. Notice that the invariant includes the invariant of the initial machine: as previously we have implicit DIST and OBST, equal to abstract ones and thus respecting abstract invariant $Prop(OBST,DIST)$. The new local part of the invariant ($Inv(OBST,DIST,obst,dist)$) expresses the parallelism between real data and detected ones. It is an example of non-trivial linking relation between abstract and concrete state. It illustrate that such relations must be well understood and thus checked to ensure the reliability of the model. In this case, $Inv(OBST,DIST,obst,dist)$ says that sensors are reliable which is a strong assertion. This may be restrained to a sub-part of sensors by further refinements, but in all cases, it will not disappear as it is a property concerning a relation between the implemented system and the real world. Thus it cannot be a proved internal consistency property of the system guaranteed by the rigorous development. Such a guarantee could be ensured by a contract with the provider of the sensors, for example. Although the method cannot prove everything, it makes hypotheses explicitly appear. In particular relations to real world necessary causes hypotheses which can be identified by the presence of real world variables in formula. It provides support for methodology to complete the method, as discussed in next section.

These refinement step examples show how to refine a state and how to refine an event by adding precision to abstract representations. There are two other important aspects we present more briefly: splitting an event into a sequence of smaller example, and decomposing a specification. In further refinement, the `move` event may be split into a `detect` event followed by a more concrete `move` event refining the abstract one. For this, `detect` should update new variables `d_obst` and `d_dist` (new events cannot modify abstract state) and `move` use the

values of these variables to update `obst` and `dist`. A detected Boolean flag, raised by `detect` and lowed by `move` can ensure the strict alternating of both events. And of course events must be carefully written in order to respect abstract behaviors and properties.

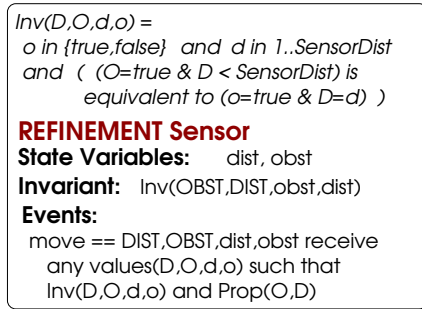


Figure 11

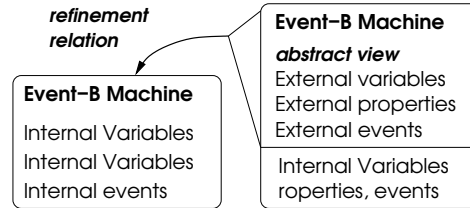


Figure 12

To complete the process we can also decompose a specification following the principles illustrated on figure 12. A part of the state of the machine is considered as “external”, which implies that the machine can’t modify it, but it can see it. Thus neither external events nor external properties contains internal variable: they represent the view the machine has of this external part. This external part may be implemented by any machine refining it, which can be developed without knowledge of the seeing machine. The view of the seeing machine should represent the minimal required properties for the machine to “work well”, i.e. technically to make its proof obligations true. One interesting application of this is to separate system and real world variable, and thus descriptions. Another most general view of this process is that it is a way to characterize suitable decomposition for horizontal structure. Indeed, as said previously, combining component with shared state may break properties. By this approach, we associate to each component the conditions under which composition preserves guaranteed properties.

All the presented techniques are tools to deal with complexity while developing or analyzing systems. Their use require some expertise to be efficient as choices are often subtle. Thus to make them more usable in practice, associated methodologies are welcome.

5 Discussion about Methodological Aspects and Conclusion

Although the B method and its variant Event-B is one of the probably the most complete fully formalized refinement process, there are similar concepts and techniques associated with other approaches. Lots of design environment and actually implemented tools exists, making use of formal methods feasible (for examples, see [19, 20, 21]). Formal models not only offer proof approaches (proof is often demanding too much) but also complementary validation tools such as fast simulation, prototyping, test generation, etc taking benefit of rigor and abstraction. As the discipline is quite recent, lots of works around this subject are academic and tools are not always easy to use for non experts, but this is evolving and the scope of the use of these methods extends. For example, US navy researchers recently developed a 3D simulator for formal models in order to work on drones ([22]). After this short overview of the current situation, this section discusses three important issues about methodologies for formal methods and their extension to other domains than computer systems.

A first aspect is the interest of introducing them early in design processes. Experience shows that simply have a rigorous non ambiguous description of high level requirement strongly enhances the development by eliminating errors dues to misunderstanding at the beginning of the process. High level models are simple which makes formal verifying efficient and useful to help debugging and implementation choices. Another aspect is that applying formal method after development supposes reverse engineering on detailed implementation, what is difficult, expensive, error prone and thus difficult to make profitable. Such low level implementation is generally complex with lots of aspects merged. Information to re-structure them is generally lost. Indeed, the product of a design process generally does not memorize inside itself all the human thinking that has structured its development. Thus efficient reverse engineering supports few automation and requires a lot of human contribution to recover or reinvent conceptual structure which is useful for efficient reasoning. Moreover, the experience of formal proof and formal refinement shows that not any structuring is suitable for feasible reasoning. Thus providing a framework based on formal refinement and decomposition guidelines to control the structuring activity during the design process is necessary to strongly master the safety and reliability of resulting complex systems. A strong dose of top-down formal development is a good solution with respect to this objective. Such an approach can offer good decompositions and components provided with rigorously specified conditions of use (and then re-use). Unfortunately, there are two major obstacles to such approaches. First it is not directly compatible with inherited design processes and expertise of current manpower. In the same way, tools only begin to become mature. It is evolving. The second obstacle is that the wanted kind of structuring is difficult and demands a strong human effort. It is perhaps also a question of maturity as structuring problem for understanding and mastering them is not generally the same as decomposing them to

distribute work or representing real world separate components. Further research, experience and maturity may lead to progress to better distinguish and combine these different aspects. Lots of proofs are not hard within the meaning of mathematical hardness but only complex in term of size, interwoven aspects and indirections due to coding and reformulating. The most the proofs are direct without non-relevant details, the most they are simple. Finally, notice that despite automatic reverse engineering cannot be considered as a panacea, abstraction and simplification techniques afford strong gains of efficiency, in particular in the domain of model checking approaches.

A second aspect is the weight of expertise, both in formal techniques and in application domain. Not considering communication between these two type of expert is quite welcome. Although some provider of formal methods are proud of offering “press-button” proofs or verifying tools (which is of course very interesting), it is not possible to avoid human expertise for an efficient using. Indeed the way models or specifications are written or structured has a strong impact on the amount of states of the machines or on the complexity of proofs to do. This originates in fundamental aspects as explained in previous paragraph, and also in specific technical choices in tools algorithms. Moreover, mathematical semantics may be subtle and the relation between models and real systems must be well understood to know what is actually proven. Thus a deep knowledge of formal semantics by formal expert avoids misunderstanding of formal model, and expertise of domain experts avoids misunderstanding of real problem or real system. In addition for combining these expertise, experts must communicate. For this, they need a common support for what the compilation of specifications is a good candidate. It is a widespread belief that formal specifications are very hard to read for non specialist. But lots of graphical presentations have been developed to make it easier. Moreover, the logic itself has originally been developed to translate “natural language” in rigorous one. Thus, provided that specifications are simple and formulas are not too long (which is the result of structuring), many people can easily become familiar with their reading. Some specification languages (such as Event-B for example) are provided with tools generating pseudo-natural language from specification, by replacing symbols by their “natural” interpretation. A last remark is that specification validation may require experts of several domains. When developing a drone, the expert of the domain in which the drone will be used ensures that the drone be able to avoid obstacle because this property is essential. But another expert might be involved to ensure that sensors are reliable (required hypothesis identified during refinement). And other experts might validate required properties about different materials and components chosen to implement the drone.

The last aspects concerns what the methodology can offer. We have presented multiple facets of the design problem. A methodology offers support to combine these facets. Guidelines for structuring are expected as the hardness of this activity is one of the main obstacles to the use of presented approaches in industry. A precise description of actors and their roles is also useful. For example a validation process can be defined for specification using tools that automatically extract properties that have to be checked by experts from specification. These properties can be classified. As formal models support different tools, methodologies can define a mapping of techniques applied to tasks. It can be configurable, and for example can allow choosing the compromise between proving and non-proving techniques depending on cost, time to spend and expected level of reliability. Software supports for such methodologies already exist for hardware/software system design. They offer specialized interfaces to handle models that are a shared reference to the different actors of the process, based on a shared vocabulary. Specialization may not only concern computer science. To achieve this presentation, we present two examples of extension of formal methodologies we are currently interested in (in order to address the risk field [23]).

- *Intrinsic Safety* is an approach that could be resumed by “secure state by default”. For example, the brakes of a train are always tight, except on command when the train is running. More generally the idea is “to get out of a safe intrinsic state only under particular conditions” is used to develop robustness against unpredicted situations. Implementation of this idea currently relies on “know-how” and physical laws. Modeling implicit hypothesis and implicitly applied rules could lead to propose a proved methodology for it, using existing theories such as Event-B in a specialized way.
- *Discretisation* is underlying state machines and semantics of temporal logics. Computer systems are mostly naturally discrete thus discrete representation does not often lead to errors. This is not the case for systems in general. For example, a drone moves in a continuous way and the model presented previously in the paper relies on a kind of regular sampling. Modeling lots of physical phenomena requires this kind of reduction and errors can result from an abusive reduction. For example there should be a minimal sampling frequency for a modeling to be reliable for some safety property. Thus it would be interesting to propose some general methodologies to handle this kind of question and reduce as much as possible the field of possible errors.

To conclude, like general knowledge about structuring reasoning, mathematical results issued from researches on design methodologies may have interest beyond the design problem and in particular for system analysis.

6 References

- [1] J. Fitzgerald, C. Gamble, P. G. Larsen, K. Pierce, and J. Woodcock, “Cyber-physical systems design: Formal foundations, methods and integrated tool chains,” in *FormaliSE 2015. Florence, Italy, May 18, 2015*, pp. 40–46, IEEE, 2015.
- [2] N. Bourbaki, “Elements de mathématiques,” 1939-Today (since 2006: Springer Verlag).
- [3] A. N. Whitehead and B. Russell, “Principia mathematica,” 1st ed. 1910-1913, 2nd ed. 1925 - 1927.
- [4] R. Carnap, “Der logische aufbau der welt,” 1928.
- [5] W. V. O. Quine, “Set theory and its logic.” Cambridge (Mass.), Harvard University Press (1969), 1963.
- [6] K. Gödel, “Über formal unentscheidbare sätze der principia mathematica und verwandter systeme,” 1931.
- [7] J. A. Goguen and R. M. Burstall, *Introducing institutions*, pp. 221–256. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984.
- [8] F. Durán and J. Meseguer, “Structured theories and institutions,” *Electronic Notes in Theoretical Computer Science*, vol. 29, pp. 23 – 41, 1999. {CTCS} ’99, Conference on Category Theory and Computer Science.
- [9] R. Diaconescu, *From Universal Logic to Computer Science, and Back*, pp. 1–16. Cham: Springer International Publishing, 2014.
- [10] S. Jacobs and V. Kuncak, *Towards Complete Reasoning about Axiomatic Specifications*, pp. 278–293. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.
- [11] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
- [12] O. Grumberg, *Abstractions and Reductions in Model Checking*, pp. 289–321. Dordrecht: Springer Netherlands, 2002.
- [13] R.-J. J. Back, A. Akademi, and J. V. Wright, *Refinement Calculus: A Systematic Introduction*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1st ed., 1998.
- [14] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki, “Casl: the common algebraic specification language,” *Theoretical Computer Science*, vol. 286, no. 2, pp. 153 – 196, 2002. Current trends in Algebraic Development Techniques.
- [15] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 1st ed., 2010.
- [16] J.-R. Abrial, *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996.
- [17] T. J. Tanzi and F. Lefeuve, “The Contribution Of Radio Sciences to Disaster Management,” in *International Symposium on Geo-information for Disaster Management - Gi4DM 2011*, May 2011,, Antalya, Turkey.
- [18] T. Tanzi and F. Delmer, *Ingénierie du risque*. Collection Sciences et technologies, Hermes Science Publications, 350p., 2006.
- [19] D. Knorreck, L. Apvrille, and R. Pacalet, “Formal system-level design space exploration,” *Concurrency and Computation: Practice and Experience*, vol. 25, no. 2, pp. 250–264, 2013.
- [20] R. Drechsler and U. Khne, *Formal Modeling and Verification of Cyber-Physical Systems: 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*. Springer Vieweg, 2015.
- [21] A. Sangiovanni-Vincentelli, “Quo vadis sld: Reasoning about trends and challenges of system-level design,” *Proceedings of the IEEE*, vol. 95, pp. 467–506, March 2007.
- [22] C. L. Heitmeyer and E. I. Leonard, “Obtaining trust in autonomous systems: Tools for formal model synthesis and validation,” in *FormaliSE 2015*, (Piscataway, NJ, USA), pp. 54–60, IEEE Press, 2015.
- [23] S. Coudert and T. J. Tanzi, “Formal methods for safe design of autonomous systems dedicated to risk management,” *First IFIP Conference on Information Technology in Disaster Risk Reduction (ITDRR 2016)*, November 16 - 18th, Sofia, Bulgaria, to appear.