



FAULTLESS SYSTEMS: YES WE CAN!

Jean-Raymond Abrial, *Swiss Federal Institute of Technology, Zurich*

Gradually introducing some simple features will eventually result in a global improvement in the software development situation.

The title of my article is intended to be provocative. We all know that faultless systems are impossible; otherwise, we'd already be constructing them. And what is a "fault"?

You probably think by now that this is yet another guru trying to sell you his latest universal panacea. Rest assured, my intention is just to remind you of a few simple facts and ideas that you might wish to use.

Faced with a terrible situation—and yes, the situation facing software and system developers is indeed terrible—we might decide to change things in a brutal way, but it never works. My philosophy is to gradually introduce some simple features that together will eventually result in a global improvement of the situation.

DEFINITIONS AND REQUIREMENTS DOCUMENT

To build correct systems, we must carefully define the way in which we judge correctness. This is the purpose of a definitions and requirements document, which must be carefully written before embarking on any system development.

It is my experience that requirements documents used in industry typically are very poor: It is often hard to understand what the requirements are and extract them from these documents. People too often justify the appropriateness of their requirements documents by the fact that they use some (expensive) tools.

This document should include two kinds of texts embedded in each other: the explanatory text and the reference text. The former contains explanations needed to understand the problem at hand. Such explanations are supposed to help a reader who encounters this problem for the first time. The latter contains definitions and requirements mainly in the form of short natural-language statements that are labeled and numbered. These definitions and requirements must be self-contained and easily separated from the accompanying explanations. They form the unique reference to correctness.

The definitions and requirements document is analogous to a book of mathematics where fragments of the explanatory text, in which the author informally explains his approach and sometimes gives some historical background, are intermixed with fragments of more formal items: definitions, lemmas, and theorems, all of which form the reference text and can easily be separated from the rest of the book.

In the case of systems engineering, we label our reference definitions and requirements along two axes.

The first axis contains the purpose (functional, equipment, safety, physical units, degraded modes, errors, and so forth) and must be defined carefully before embarking on writing the definition and requirements document since it may vary from one project to the other. Note that the "functional" label corresponds to requirements dealing with the intended software's specific task, whereas the "equipment" label deals with *assumptions* (which we also call requirements) that must be guaranteed concerning the environment in which the intended software is situated.

The second axis places the reference items within a hierarchy going from very general (abstract) definitions or requirements down to increasingly more specific ones

imposed by system promoters. It is very important that the stakeholders must agree upon and sign off on this rewriting of the definition and requirements document.

At the end of this phase, however, we have no guarantee that the desired properties of our system that have been written down can indeed be fulfilled: Mandating that an airplane must fly doesn't mean that it actually will. However, quite often after the writing of such a document, people rush into the programming phase, and we know very well what the outcome is. What is needed is to undertake an intermediate phase before *programming*.

MODELING VERSUS PROGRAMMING

Programming is the activity of constructing a piece of formal text that is supposed to instruct a computer in how to fulfill certain tasks. Our intention is not to do that.

Our task is not limited to the software part alone, because what we intend to build is a system within which the piece of software we will construct is just one component among many others. In doing this as engineers, we are not supposed to instruct a computer; rather, we are supposed to instruct ourselves. To do this in a rigorous way, we have no choice but to perform a complete *modeling* of our future system, including the software that will eventually be constructed and its environment—which includes equipment, physically varying phenomena, other software, and even users.

Programming languages are of no help in doing this. All this must be carefully modeled so that we know the exact assumptions within which our software is to behave. Modeling is therefore the main task of system engineers. Programming is then merely a subtask that may very well be performed automatically.

Computerized system modeling had been undertaken in the past (and still is) with the help of simulation languages such as Simula 67, the predecessor of all object-oriented programming languages. What we propose here is also to perform a simulation. But rather than doing it with the help of a simulation language for which the outcome can be inspected and analyzed, we propose to do it by constructing *mathematical models* that will be analyzed by doing *proofs*. Physicists or operational researchers proceed in this way. We'll do the same.

Since we are not instructing a computer, we do not have to say what is to be done; rather, we need to explain and formalize what we can *observe*. This immediately raises the question, How can we observe something that does not exist yet? Simple: It doesn't exist yet in the physical world but, for sure, it exists in our mind.

Engineers or architects always proceed in this way: They construct artifacts according to the predefined representation they have of them in their mind.

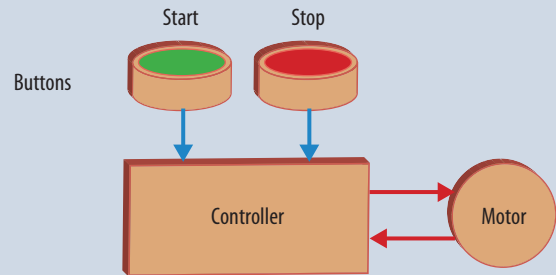


Figure 1. Three communicating discrete transition systems.

DISCRETE TRANSITION SYSTEMS AND PROOFS

In addition to formalizing our mental representation of the future system, modeling also consists of proving that this representation fulfills certain desired properties, namely those stated informally in the definition and requirements document.

To perform this joint task of simulation and proof, we use a simple formalism—*discrete transition systems*. In other words, whatever the modeling task we have to perform, we always represent the components of our future systems by means of a succession of states intermixed with sudden transitions, also called events.

From the modeling point of view, it is important to understand that there is no fundamental difference between a human pressing a button, a motor starting or stopping, or a piece of software executing certain tasks, all of them being situated within the same global system. Each of these activities is a discrete transition system working on its own and communicating with others as shown in Figure 1. They are embarked upon together in the distributed activities of the system as a whole. This is the way we would like to perform our modeling task.

It happens that this very simple paradigm is extremely convenient. In particular, the *proving* task is partially performed by demonstrating that the transitions of each component preserve several desired global properties that the states of our components must permanently obey. These properties are the so-called invariants, which most often are transversal properties involving the states of multiple system components. The corresponding proofs are called the *invariant preservation proofs*.

States and events

Roughly speaking, several variables define a *state* (as in an imperative program). However, the difference is that these variables might be any integers, pairs, sets, relations, functions, and so on—that is, any mathematical object that can be represented in set theory—not just computer objects (limited integer and floating-point numbers, arrays, files, and the like). Apart from the variables definitions,

we might have invariant statements, which can be any predicate expressed within the notation of first-order logic and set theory. By putting all this together, a state can be simply abstracted to a set.

An event can be abstracted to a simple binary relation built on the state set. This relation connects two successive states, considered just before and just after the event “execution.” However, defining an event directly as a binary relation would not be very convenient. A better notation involves splitting an event into two parts: *guards* and *actions*.

A guard is a predicate, and all the guards conjoined in an event form the corresponding relation’s domain. An action is a simple assignment to a state variable. The actions of an event are supposed to be “executed” simultaneously on different variables. Variables that are not assigned are unchanged. This is all the notation we need to define our transition systems.

Modeling a large system containing many discrete transition components must be undertaken in successive steps.

Horizontal refinement and proofs

Modeling a large system containing many discrete transition components must be undertaken in successive steps. Each step makes the model richer by first creating and then enriching the states and transitions of its various components, first in a very abstract way and later by introducing more concrete elements. This activity is termed *horizontal refinement* (or superposition).

In doing this, the system engineer explores the definition and requirements document and gradually extracts from it some elements to be formalized. Thus, the *traceability* of the definitions and requirements starts within the model. If we discover through modeling that the definition and requirements document is incomplete or inconsistent, we then have to edit it accordingly.

By applying this horizontal refinement approach, we have to perform some proofs—namely that a more concrete refinement step does not invalidate what has been done in a more abstract step. These are the refinement proofs. Note, finally, that the horizontal refinement steps are complete when there are no remaining definitions and requirements that have not been taken into account in the model.

In undertaking horizontal refinement, we do not care about implementability. Our mathematical model is built using the set-theoretic notation to write down the state invariants and the transitions. When undertaking horizontal refinement, we extend a model’s state by adding new variables. We can strengthen an event’s guards or add new guards. We also add new actions in an event. Finally, it is possible to add new events.

Vertical refinement and proofs

A second kind of refinement takes place when all horizontal refinement steps have been performed. As a result, we do not enter any more new details of the problem in the model; rather, we transform some states and transitions of our discrete system so that it can easily be implemented on a computer. This is called *vertical refinement* (or data refinement), which often can be performed by a semiautomatic tool. *Refinement proofs* must also be performed to make sure that our implementation *choice* is coherent with the more abstract view.

A typical example of vertical refinement is the transformation of finite sets into Boolean arrays together with the corresponding transformations of set-theoretic operations (union, intersection, inclusion, and so on) into program loops. When conducting a vertical refinement, we can remove some variables and add new ones. An important aspect of vertical refinement is the so-called “gluing” invariant linking the concrete and abstract states.

Communication and proofs

An important aspect of the modeling task is concerned with the communication between the future system’s various components. We must be very careful here to proceed again by successive refinements. It is a mistake to immediately model the communication between components as they will appear in the final system.

A good approach is to consider that each component has the “right” to directly access the state of other components (which are still very abstract too). In doing that, we “cheat,” of course, as this is clearly not the way things work in reality. But this is convenient to use in the initial horizontal refinement steps as our components are gradually refined, with their *communication* also becoming gradually richer as we go down the refinement steps. Only at the end of the horizontal refinement steps is it appropriate to introduce various channels corresponding to the real communication schemes at work between components and to possibly decompose our global system into several communicating subsystems.

We will then notice that each component reacts to the transitions of others with a fuzzy picture of their states because it takes time to transmit the messages between the components. We then have to prove that despite this time shift, things remain “as if” such a shift did not exist. This is yet another refinement proof that we have to perform.

BEING FAULTLESS: WHAT DOES THAT MEAN?

We are now ready to be precise about what we mean by a “faultless” system, achieving which represents our ultimate goal, as the title of this article suggests.

Let’s consider a program controlling a train network. If it is not developed to be correct by construction, then after writing it we can certainly never prove that this program

will guarantee that two trains never collide. It is too late. The only thing we might sometimes, and unfortunately not always, be able to test or prove is that such a program does not include array accesses that are out of bounds or dangerous null pointers that might be accessed, or that it does not contain the risk of some arithmetic overflow. However, recall that it was precisely this undetected problem that caused the Ariane 5 crash on its maiden voyage.

There is an important difference between a solution validation and a problem validation. It seems that there is a lot of confusion here as people do not make any clear distinction between the two. A solution validation is concerned solely with the constructed software, and it validates this piece of code against several software properties. Conversely, a problem validation is concerned with the system's overall purpose—for example, to ensure that trains travel safely within a given network. To do this, we must prove that all components of this system (not just the software) harmoniously participate in this global goal.

To prove that our program will guarantee that two trains will never collide, we must construct this program by modeling the problem. And, of course, the property in question must be part of the model to begin with. We should note, however, that people sometimes succeed in doing some sort of problem proofs directly on the solution (the program). This is done by incorporating some so-called “ghost” variables dealing with the problem inside the program. Such variables are then removed in the final code. We consider this approach a rather artificial afterthought.


During the horizontal refinement phase of our model development, we shall take account of many properties. At the end of the horizontal refinement phase, we shall then be able to know exactly what we mean by this noncollision property. In doing so, we shall make all the assumptions precise (in particular any environmental assumptions) under which our model will guarantee that two trains will never collide.

As can be seen, the property alone is not sufficient. By exhibiting all these assumptions, we are doing a problem validation that is completely different in nature than the one we can perform on the software only. Using this kind of approach for all properties of our system will allow us to claim that, at the end of our development, our system is faultless by construction. As such, we have made very precise what “faults” are under consideration and, in particular, their relevant assumptions.

We should note a delicate point here. We pretended that this approach allows us to produce a final version of the software that is correct by construction relative to its surrounding environment. In other words, the global system is faultless. We achieved this by means of proofs performed during the modeling phase, in which we constructed a model of the environment. We said earlier that this environment was made up of equipment, physical phenomena,

pieces of software, and users. It is quite clear that these elements cannot be modeled completely. Rather than saying that our software is correct relative to its environment, it would be more appropriate to be modest and say that our software is correct relative to the model of the environment we have constructed. This model is certainly only an approximation of the physical environment. Should this approximation be too far from the real environment, then it will still be possible for our software to fail under unforeseen external circumstances.

In short, we can only pretend to achieve a relative faultless construction, not an absolute one, which is clearly impossible. A problem solution for which is still in its infancy is finding the right methodology to perform an environmental model that is a “good” approximation of the real environment. It is clear that a probabilistic approach would certainly be very useful for doing this.



We can only pretend to achieve a relative faultless construction, not an absolute one, which is clearly impossible.

About proofs

Clearly, we need a tool that automatically generates the proofs we perform during the modeling process since it would be foolish and error prone to let a human write the formal statements for thousands of such proofs. As a rule of thumb, we want a tool that will automatically discharge 90 percent of the proofs.

An interesting question is then to study what happens when an automatic proof fails. It might be because the automatic prover is not smart enough, the statement we are trying to prove is false, or the statement to be proven just cannot be proved.

In the first case, we must perform an interactive proof. In the second, the model must be modified significantly. In the last case, the model must be enriched. The last two cases are very interesting as the proof activity plays the same role for models as the one played by testing for programs. The final percentage of proofs discharged automatically is a good indication of the quality of the model. If there are too many interactive proofs, this may signify that the model is too complicated. By simplifying the model, we often also significantly augment the percentage of automatically discharged proofs.

Design patterns

Design patterns became very popular some years ago in object-oriented software development. But the idea is more general than that: It can be fruitfully extended to any particular engineering discipline and, in particular, to system engineering as envisaged here.

The idea is to write down some predefined little engineering recipes that can be reused in many different situations provided that these recipes are instantiated accordingly. In our situation, it takes the form of some proven, parameterized models that can be incorporated in a large project. The nice effect is that it saves redoing proofs that have already been done in the pattern development. Tools can be developed to easily instantiate and incorporate patterns in a systematic fashion.

The idea is to use animation as early as possible during the horizontal refinement phase, even on very abstract steps.

Animation

Here is a strange thing: Thus far we have strongly proposed to base our correctness assurance on modeling and proof. Here, we are going to say that, well, it might also be good to animate—that is, execute—our models. But, we thought that mathematics was sufficient and precise and that there was no need to execute. Is there any contradiction here? Are we in fact not so sure after all that our mathematical treatment was sufficient, that mathematics are always “true”? No. After a proof of the Pythagorean theorem, no mathematician would think of measuring the hypotenuse and the two legs of a right triangle to check the validity of the theorem. So why would we execute our models?

We have certainly proved something, and we have no doubts about our proofs. But are we sure that what we proved was indeed the right thing to prove? This may be a bitter pill to swallow: We painfully wrote the definition and requirements document precisely for that reason—to know exactly what we have to prove. And now we claim that perhaps what the requirements document said was not what is wanted.

Directly animating the model—we are not talking here about a special simulation, but are still using the very model that we proved. Showing this animation of the entire system (not only the software part) on a screen is a useful means of checking in another way (besides referencing the requirements document) that what we want is indeed what we wrote. Quite often, by doing this, we discover that our requirements document was not accurate enough, or that it required properties that are not included, or even properties that are different from what we want.

Animation complements modeling. It allows us to discover that we might have to change our minds very early on. The interesting thing is that it does not cost that much money, far less indeed than doing a real execution on the final system and discovering far too late that the system we built is not the system we want.

It may seem that animation must be performed after proving (as an additional phase before programming). But in fact, the idea is to use animation as early as possible during the horizontal refinement phase, even on very abstract steps. The reason is that if we have to change our requirements and thus redo some proofs, we must know exactly what we can save in our model and where we have to modify the construction.

There is another positive outcome as a result of animating and proving simultaneously. Recall that we said that proving was a way to debug our model: A proof that cannot be done is an indication that there is a “bug” in our model or that it’s a poor model. The fact that an invariant preservation proof cannot be done can be pointed out and explained by an animation even before doing the proof. Animation often easily discovers deadlock freedom counterexamples.

Note that animation does not mean that we can suspend our proof activity, but it is a very useful complement to it.

Tools

Tools are important for developing correct systems. Here we propose to depart from the usual approach in which there is a formal text file containing models and their successive refinement. It is far more appropriate to have a database at our disposal. This database handles objects such as models, variables, invariants, events, guards, actions, and their relationships.

Static analyzers, which are widely available, can be used on these components for lexical analysis, name clash detection, mathematical text syntactic analysis, refinement rules verification, and so on. An important tool is the proof obligation generator, which analyzes the models (invariants, events) and their refinements to produce corresponding statements to prove.

Finally, we need some automatic and interactive proving tools to discharge the proof obligations provided by the previous tool. An important thing to understand here is that the proofs to be performed are not the kind of proofs a professional mathematician would do or be interested in. Our proving tool must take this into account.

In a mathematical project, the mathematician is interested in proving one theorem (say, the four-color theorem) together with some lemmas (say, 20 of them). The mathematician does not use mathematics to accompany the construction of an artifact. During the mathematical project, the problem does not change as this is still the four-color problem.

In an engineering project, thousands of predicates must be proved. Moreover, what we have to prove is not known right from the beginning. Again, we do not prove that trains do not collide: We prove that the system we are constructing ensures that, under certain hypotheses about the environment, trains do not collide. What we have to prove evolves with our understanding of the problem and

our (nonlinear) progress in the construction process.

As a consequence, an engineering prover needs to have functionality that is not necessarily needed in provers dedicated to performing proofs for mathematicians. Two of these functionalities are differential proving (how to figure out which proofs have to be redone when we make a slight modification to our model) and proving in the presence of useless hypotheses.

To the tools we have already mentioned, it is useful to add several other tools using the same core database, tools for animation, model-checking, UML transformation, design patterns, composition, decompositions, and so on. It means that our tooling system must be built in such a way that this extension approach is facilitated. A tool developed according to this philosophy is the Rodin platform, which can be freely downloaded from www.event-b.org.

Legacy code

When dealing with legacy code, we either want to develop a new piece of software that is connected to some legacy code or renovate particular legacy code.

The first and most common approach is usually found in the development of a new piece of software. In this case, the legacy code is just an element of our new product's environment. The challenge is to capture legacy code behavior so that we can enter it in the model as we do with any other element of the environment. To do this, our new product's requirements document must contain some elements concerned with the legacy code. Such requirements (assumptions) must be defined informally. The goal is to develop in our model the minimal interface compatible with the legacy code. As usual, the key is abstraction and refinement: How can we gradually introduce the legacy code into our model in such a way that we take full account of the concrete interface it offers?

The second problem is far more difficult. In fact, such renovations often give very disappointing results. People tend to consider that the legacy code "is" the requirements document of the renovation. This is an error.

The first step is to write a new requirements document, not hesitating to deviate completely from the legacy code and define abstract requirements that are independent from the precise implementation seen in the legacy code.

The second step is to renovate the legacy code by developing and proving a model of it. The danger here is that we try to mimic the legacy code too closely because it might contain aspects that are not comprehensible (except by the absent legacy code programmers) and that are certainly not the result of a formal modeling approach.

Our advice here is to think twice before embarking on such a light renovation. A better approach is to develop a new product. People think it might consume more time and money than a simple renovation; experience shows that it is rarely the case.

Set-theoretic notation

Physicists or operational researchers, who also proceed by constructing models, never invented specific languages to do so: They all use classical set-theoretic notations.

Computer scientists, because they have been educated to program only, believe that it is necessary to invent specific languages to do the modeling. This is an error. Set-theoretic notations are well suited to performing system modeling; moreover, we can understand what it means when we write a formal statement.

We also frequently hear that we must hide the usage of mathematical notation because engineers will not understand it or will be afraid of it. This is nonsense. Is it necessary to hide the mathematical notation used in the design of an electrical network because electrical engineers would be afraid of it?



The challenge is to capture legacy code behavior so that we can enter it in the model as we do with any other element of the environment.

Other validation approaches

For decades, various approaches have been dedicated to the validation of software. Among them are tests, abstract interpretation, and model checking.

These approaches validate the solution, the software—not the problem, the global system. In each case, we construct a piece of software and then, and only then, try to validate it (although this is not entirely the case with model checking, which is also used for problem validation). To do so, we think of a certain desired property and check that indeed our software is consistent with it. If this is not the case, then we have to modify the software and thus, quite often, introduce more problems. It is also well known that such approaches are very expensive, far more so than the pure development cost.

We do not think that these approaches alone are appropriate. Of course, we are not saying that we should reject them. We are just saying they may complement the modeling and proving approach and not replace it.

INNOVATION

Big industrial corporations often cannot innovate. They do so sometimes, provided a very large amount of money is given to them precisely for this purpose. It is well known that many so-called R&D divisions of big companies are not providing any significant technologies for their business units.

Nevertheless, financing agencies still insist on having practical research proposals connected with such large companies. This is an error. They should do a better job by

accepting connections with far smaller, more innovative entities. It is my belief that the introduction into industry of the approach I advocate should be done through small innovative companies rather than big corporations.

EDUCATION

Many of the people presently involved in large software engineering projects are not correctly educated. Companies think that programming jobs can be done by junior people with little or no mathematical background and interest (quite often programmers do not like mathematics: This is why they choose computing in the first place). All this is bad. A system engineer's basic background must be a mathematical education at a good (even high) level.

Computing should come second, after the necessary mathematical background has been well understood. As long as this is not the case, things cannot improve. Of course, it is clear that many academics will disagree with this: It is not the smallest problem we have to face. Quite often, academics confuse computation and mathematics.

It is far less expensive to have a few well-educated people than an army of people who are not educated at the right level. This is not an elitist attitude: Who would think that a doctor or an architect can perform well without a proper education in a specific discipline? Again, the basic discipline of system and software engineers is (discrete) mathematics.

Two specific topics to be taught to future software engineers are the writing of requirements documents (this is barely present in practical software engineering curricula) and the construction of mathematical models. Here the basic approach is a practical one: These topics must be taught by having the students explore many examples and projects. Experience shows that mastering the mathematical approach (including the proofs) is not a problem for students with a good mathematical background.

TECHNOLOGY TRANSFER

Technology transfer of this kind in industry is a serious problem due to the extreme reluctance of managers to modify their development process. Usually such processes are difficult to define and more difficult to put into practice, which is why managers do not like to modify them.

The incorporation of an important initial phase of requirements document writing followed by another important phase of modeling is usually regarded as dangerous, as these additional phases impose some significant expenses at the beginning of a project. However, experience shows that the overall expenditure is drastically decreased since the very costly testing phase at the end can be significantly reduced, as well as the considerable efforts needed to patch design errors.

But above all, the initial action to be done to transfer a technology to industry is to perform a significant preliminary education effort. Without that initial effort, any technology transfer attempt is certain to fail.

The ideas presented in this article are not new. Most of them come from the seminal ideas of action systems developed in the 1980s and 1990s, including those by Ralph-Johan Back and Reino Kurki-Suonio¹ and Michael Butler.² More recently, some of the ideas presented in this article have been put into practice (www.event-b.org).³

The simple ideas presented here offer suggestions for how to improve the situation of computerized system development. Now the question is clearly, Has all this been put into practice? The answer is a small yes: Faultless systems exist.^{4,5} However, many more steps must be performed to have these ideas more widely understood and accepted. This is what we are presently doing in the European Project Deploy (www.deploy-project.eu). **□**

Acknowledgments

I would like to thank Michael Butler for many discussions concerning these matters. I warmly thank other people who sent me very interesting comments on earlier drafts of this article: David Basin, Egon Boerger, Mike Hinchey, Tony Hoare, Michael Jackson, Peter Mueller, Michel Sintzoff, Bernard Sufrin, and Pamela Zave.

References

1. R-J. Back and R. Kurki-Suonio, "Decentralization of Process Nets with Centralized Control," *Proc. 2nd ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, ACM Press, 1983, pp. 131-142.
2. M. Butler, "Stepwise Refinement of Communicating Systems," *Science of Computer Programming*, vol. 27, no. 2, 1996, pp. 139-173.
3. J-R. Abrial, *Modelling in Event-B: System and Software Engineering*, Cambridge University Press, to be published, 2009.
4. F. Badeau, "Using B as a High Level Programming Language in an Industrial Project: Roissy VAL," *Proc. 4th Int'l Conf. B and Z Users (ZB 05)*, LNCS 3455, Springer Verlag, 2005, pp. 334-354.
5. P. Behm, "Meteor: A Successful Application of B in a Large Project," *Proc. World Congress on Formal Methods in the Development of Computing Systems (FM 99)*, LNCS 1708, Springer, 1999, pp. 369-387.

Jean-Raymond Abrial was a senior researcher and guest professor in the Department of Information Security, Swiss Federal Institute of Technology, Zurich, Switzerland. His research interests are the application of rigorous approaches in system development, the usage of refinement, and that of proofs. He graduated from Ecole Polytechnique in France with a strong emphasis on mathematics. Contact him at [jrabil@neuf.fr](mailto:jrabrial@neuf.fr).