Faultless Systems: Yes We Can!

This title is certainly provocative. Everyone knows that this claim corresponds to something that is impossible. No! One can not construct faultless systems, just have a look around. Should this have been possible, it would have been already done for a long time. And to begin with: what is a "fault"?

So, how can someone imagine the contrary? You might think: yet another guru trying to sell us his latest Universal Panacea. Dear reader, be reassured, this prologue does not contain any new bright solutions and moreover it is not technical: you'll have no complicated concepts to swallow. My intention is just to remind you of a few simple facts and ideas that you might use if you wish to do so.

I would like to play the role of those who are faced with a terrible situation (yes, the situation of computerized system developments is not far from being terrible: as a measurement, just consider the money thrown out of the window with systems that fail). Faced with a terrible situation, one might decide to change things in a brutal way: it never works. Another approach is to gradually introduce some simple features which *together* will eventually result in a global improvement of the situation. The latter is my philosophy.

Definition and Requirement Document

Since it is our intention to build correct systems, we need first to carefully define the way we can judge that this is indeed the case. This is the purpose of a "Definitions and Requirement" document, which has to be carefully written before embarking in any computerized system development.

But you can say that lots of industries have such documents: they already exist, so why bother? Well, it is my experience that most of the time, requirement documents that are used in industry are *very poor*: it is often very hard just to understand what the requirements are and thus to extract them from these documents. People too often justify the appropriateness of their requirement documents by the fact that they use some (expensive) tools!

I strongly recommend to rewrite requirement documents along the simple lines presented in this section.

This document should be made of 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 and need some elementary explanations. The latter contains definitions and requirements mainly in the form of short natural language statements that are labelled and numbered. Such definitions and requirements are dryer than the accompanying explanations. However, they must be self-contained and thus constitute the unique reference for correctness.

The definition and requirement document bears an analogy with a book of mathematics where fragments of the *explanation* text (where the author explains informally 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 system engineering, we label our reference definitions and requirements along two axis. The first one contains the purpose (functional, equipment, safety, physical units, degraded modes, errors,...) while the second one contains the abstraction level (high, intermediate, low, ...).

The first axis must be defined carefully before embarking in the writing of the definition and requirement document since it might be different from one project to the other. Note that the "functional" label corresponds to requirements dealing with the specific task of the intended software, whereas the "equipment" label deals with *assumptions* (which we also call requirements) that has to be guaranteed concerning the environment situated around our intended software. Such an environment is made of some pieces of equipments, some physical varying phenomenons, other pieces of software, as well as system users. The

second axis places the reference items within a hierarchy going from very general (abstract) definitions or requirements down to more and more specific ones imposed by system promoters.

It is very important that this re-writing of the definition and requirement document must be agreed upon and signed by the stakeholders.

At the end of this phase however, we have *no guarantee* that the written desired properties of our system can indeed be fulfilled. This is not by writing that an intended airplane must fly that it indeed 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 an intermediate phase to be undertaken *before programming*: this is the purpose of what is explained in the next section.

Modelling vs. Programming

Programming is the activity of constructing a piece of formal text that is supposed to instruct a computer how to fulfil certain tasks. *Our intention is not to do that*. What we intend to build is a system within which there is a certain piece of software (the one we shall construct), which is a component among many others. This is the reason why our task is not limited to the software part only.

In doing this as engineers, we are not supposed to instruct a computer, we are rather supposed to instruct ourselves. For doing this in a rigorous way, we have no choice but to perform a complete *modelling* of our future system including the software that will be constructed eventually, as well as its environment which is made, again, of equipment, physical varying phenomenons, other software, and even users. Programming languages are of no help for doing this. All this has to be carefully modelled so as to be able to know the exact assumptions within which our software is going to behave.

Modelling is the main task of system engineers. Programming becomes then a sub-task which might very well be performed automatically.

Computerized system modelling had been done in the past (and still now) with the help of simulation languages such as SIMULA-67 (the ancestor of 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 whose outcome can be inspected and analyzed, we propose to do it by constructing *mathematical models* which will be analyzed by doing *proofs*. Physicists or operational researchers do 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, we have rather to explain and formalize what we can *observe*. But immediately comes the question: how can we observe something that does not exist yet? The answer to this question is simple: it certainly does not exist yet in the physical world but, for sure, it exists in our mind. Engineers or architects always proceed in this way: they construct artefacts according to the pre-defined representation they have of them in their mind.

Discrete Transition Systems and Proofs

As said in the previous section, modelling is not just formalizing our mental representation of the future system, it also consists in *proving* that this representation fulfils certain desired properties, namely those stated informally in the definition and requirement document briefly described above.

In order to perform this joint task of simulation and proofs, we use a simple formalism, that of *discrete transition systems*. In other words, whatever the modelling 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 point of view of modelling, it is important to understand that there is *no fundamental differences* between a human being 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. They are together embarked in the distributed activities of the system as a whole. This is the way we would like to do our modelling 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 a number of desired global properties which must be permanently obeyed by the states of our components. These properties are the so-called invariants. Most of the time, these invariants are transversal properties involving the states of multiple components in our system. The corresponding proofs are called the *invariant preservation proofs*.

States and Events

As seen in previous section, a discrete transition component is made of a state and some transitions. Let us describe this here in simple terms.

Roughly speaking, a *state* is defined (as in an imperative program) by means of a number of variables. However, the difference with a program is that these variables might be any integer, pairs, sets, relations, functions etc. (i.e. any mathematical object representable within set theory), not just computer objects (i.e. limited integer and floating numbers, arrays, files, and the like). Besides 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.

Exercises: what is the state of the discrete system of a human being able to press a button? What is the state of the discrete system of a motor able to start and stop?

Taken this into account, an *event* can be abstracted to a simple binary relation built on the state set. This relation represents the connection existing between 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 consists in splitting an event into two parts: the *guards* and the *actions*.

A guard is a predicate and all the guards conjoined together in an event form the domain of the corresponding relation. 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 are using for defining our transition systems.

Exercises: What are the events of the discrete system of a human being able to press a button? What are the events of the discrete system of a motor able to start and stop? What is the possible relationship between both these systems?

At this stage, we might be slightly embarrassed and discover that it is not so easy to answer the last question. In fact, to begin with, we have not followed our own prescriptions! Perhaps, would have it be better to first write down a definition and requirement document concerned with the user/button/motor system. In doing this, we might have discovered that this relationship between the motor and the button is not that simple after all. Here are some questions that might come up: do we need a single button or several of them (i.e. a start button, and a stop button)? Is the latter a good idea? In case of several buttons, what can we observe if the start button is pressed while the motor is already started? In this case, do we have to release the button to later re-start the motor? And so on. We could also have figured out that rather than considering separately a button system and a motor system and then *composing* them, it might have been better to consider first a single problem which might later be *decomposed* into several. Now, how about putting a piece of software between the two? And so on.

Horizontal Refinement and Proofs

The modelling of a large system containing many discrete transition components is not a task that can be done in one shot. It has to be done in successive steps. Each of these steps make 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 called *horizontal refinement* (or superposition).

In doing this, the system engineer explores the definition and requirement document and gradually extract from it some elements to be formalized: he thus starts the *traceability* of the definitions and requirements within the model. Notice that quite often he discovers by modelling that the definition and requirement document is incomplete or inconsistent: he has then 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 does not remain any definition and requirement that have not been taken into account in the model.

In doing horizontal refinement we do not care about implementability. Our mathematical model is done using the set-theoretic notation to write down the state invariants and the transitions.

When doing an horizontal refinement, we extend the state of a model by adding new variables. We can strengthen the guards of an event or add new guards. We also add new actions in an event. Finally, it is possible to add new events.

Vertical Refinement and Proofs

There exists a second kind of refinement that 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, we rather transform some state and transitions of our discrete system so that it can easily be implemented on a computer. This is called *vertical refinement* (or data refinement). It can often be performed by a semi-automatic tool. *Refinement proofs* have also to be performed in order to be 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, etc.) into program loops.

When doing 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

A very important aspect of the modelling task is concerned with the *communication* between the various components of the future system. One has to be very careful here to also proceed by successive refinements. It is a mistake to model immediately the communication between components as they will be in the final system. A good approach to this is to consider that each component has the “right" to *access directly* the state of other components (which are still very abstract too). In doing that we “cheat" of course as it is clearly not the way it works in reality. But it is a very convenient way to be used in the initial horizontal refinement steps as our components are gradually refined with their communication becoming also gradually richer as one goes 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 sub-systems.

One can then figure out that each component reacts to the transitions of others with a fuzzy picture of their states. This is because the messages between the components do take some time to travel. One has then to prove that in spite of this time shift, things remain “as if" such a shift did not exist. This is yet another *refinement proof* that one has to perform.

Being Faultless: What does it Mean?

We are now ready to make precise what we mean by a "faultless" system, which represents our ultimate goal as the title of this prologue indicates.

If a program controlling a train network is not developed to be correct by construction, then after writing it, you can certainly never prove that this program will guarantee that two trains never collide. It is too late. The only thing you might sometimes (not always unfortunately) be able to test or prove is that such a program has not got array accesses that are out of bounds, or dangerous null pointers that might be accessed, or else that it does not contain the risk of some arithmetic overflow (although, remember, this was precisely this undetected problem that caused the Ariane 5 crash in its maiden voyage).

There is an important difference between a solution validation versus a problem validation. It seems that there is a large 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 a number of *software properties* as mentioned above (out of bound array access, null pointers, overflows). On the contrary, a problem validation is concerned with the *overall purpose* of our system (i.e. to ensure that trains safely travel within a given network). For doing this, we have to prove that all components of this system (not the software only) harmoniously participate in this global goal.

For proving that our program will guarantee that two trains will never collide we have to construct this program by modelling the problem. And, of course, a significant part of this is that the property in question must be *part of the model* to begin with.

One should notice 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 on the final code. We consider that this approach is a rather artificial afterthought. The disadvantage of this approach is that it focuses attention on the software rather than on the wider problem. In fact, this use of ghost variables just highlights the need for abstraction when reasoning at the problem level. The approach advocated here is precisely to start with the abstractions, reason about those, and introduce the programs later.

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 non-collision property. In doing so, we shall make precise all *assumptions* (in particular environment 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. For this, we have made very precise what we call the “faults” under consideration (and in particular their relevant assumptions).

However, one should note a delicate point here. We pretended that this approach allows us to produce a final software that is correct by construction relative to its surrounding environment. In other words, the

global system is faultless. This has been done by means of proofs performed during the modelling phase where we constructed a model of the environment. Now we said earlier that this environment was made of equipment, physical phenomenons, pieces of software, and also users. It is quite clear that these elements cannot be formalized completely. Rather than to say that our software is correct relative to its environment, it would be more appropriate to be more modest by saying 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 be possible that our software fails under unforeseen external circumstances.

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

About Proofs

In previous sections, we mentioned several times that we have to perform proofs during the modelling process. First of all, it must be clear that we need a tool for generating automatically what we have to prove. It would be foolish (and error prone) to let a human being write down explicitly the formal statements that must be proved for the simple reason that it is common to have thousands of such proofs. Second, we also need a tool to perform the proofs automatically: a typical desirable figure here is to have 90% of the proofs being discharged automatically.

An interesting question is then to study what happens when an automatic proof fails. It might be because: (1) the automatic prover is not smart enough, or (2) the statement to prove is false, or else (3) the statement to prove cannot be proved. In case (1), we have to perform an interactive proof (see the "Tool" section below). In case (2), the model has to be significantly modified. In case (3), the model has to be enriched. Cases (2) and (3) are very interesting; they show that the proof activity plays the same role for models as the one played by testing for programs.

Also notice that the final percentage of automatic proofs is a good indication of the quality of the model. If there are too many interactive proofs it might signify that the model is too complicated. By simplifying the model we often also significantly augment the percentage of automatically discharged proofs.

Design Pattern

Design patterns have been made very popular some years ago by the book written on them for object oriented software developments [3]. 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 these recipes are instantiated accordingly. In our case, it takes the form of some proved parameterized models which can be incorporated in a large project. The nice effect is that it saves redoing proofs that have been done once and for all in the pattern development. Tools can be developed to easily instantiate and incorporate patterns in a systematic fashion.

Animation

Here is a strange thing: in previous sections we heavily proposed to base our correctness assurance on modelling and *proving*. And in this section 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 precisely 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 executing our models?

We have certainly proved something and we have no doubts about our proofs, but more simply are we sure that what we proved was indeed the right thing to prove? Things might be difficult to swallow here: we (painfully) wrote the definition and requirement document precisely for that reason, to know exactly what we have to prove. And now we claim that perhaps what the requirement document said was not what is wanted. Yes, that is the way it is: things are not working in a linear fashion.

Animating directly the model (we are not speaking here of doing a special simulation, we are using the very model which we proved) and showing this animation of the entire system (not only the software part) on a screen is very useful to check in another way (besides the requirement document) that what we want is indeed what we wrote. Quite often, by doing this, we discover that our requirement document was not accurate enough or that it required properties that are not indispensable or even different from what we want.

Animation complements modelling. It allows us to discover that we might have to change mind 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 (but far too late) that the system we built is not the system we want.

It seems that animation has to be performed after proving (as an additional phase before the programming one). No, 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) it is very important to know exactly what we can save in our model and where we have to modify our model construction.

There is another positive outcome in animating and proving simultaneously. Remember, we said that proving was a way to debug our model: a proof that cannot be done is an indication that we have a "bug" in our model or that our model is too poor. The fact that an invariant preservation proof cannot be done can be pointed out and explained by an animation even before doing the proof. Deadlock freedom counter-examples are quite often discovered very easily by animation. Notice that animation does not mean that we can suspend our proof activity, we just wanted to say that it is a very useful complement to it.

Tools

Tools are important to develop correct systems. Here we propose to depart from the usual approach where one would have a (formal) text file containing models and their successive refinement. It is far more appropriate to have a *database* at one's disposal. This database handles modelling objects such as models, variables, invariant, events, guards, actions, and their relationships as we have presented them in previous sections.

Usual *Static Analyzers* can be used on these components for lexical analysis, name clash detection, mathematical text syntactic analysis, refinement rules verification and so on.

As said above, an important tool is the one, called the *Proof Obligation Generator*, that analyzes the models (invariants, events) and their refinements in order to produce corresponding statements to prove.

Finally, some *Proving Tools* (automatic and interactive) are needed 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 (and be interested in). Our proving tool has to take this into account.

In a mathematical project, the mathematician is interested by 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 artefact. During the mathematical project, the problem does not change (this is still the four color problem).

In an engineering project, thousands of predicates have to be proved. Moreover, what we have to prove is not known right from the beginning. Note that 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 (non linear) progress in the construction process.

As a consequence, an engineering prover needs to have some functionalities which are not needed in provers dedicated to perform proofs for mathematicians. To cite two of these functionalities: differential proving (how to figure out which proofs have to be redone when a slight modification in our model occurs) and proving in the presence of useless hypotheses.

Around the tools we presented in this section, it is very useful to add a number of other tools using the same core database: animating tools, model-checking tools, UML transformation tools, design pattern tools, composition tools, decompositions tools, 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 [4].

The Problem of Legacy Code

The legacy code question has a dual aspect: either (1) we want to develop a new piece of software which is connected to some legacy code, or (2) we want to renovate a certain legacy code.

Problem (1) is the most common one: it is almost always found in the development of a new piece of software. In this case, the legacy code is just an element of the environment of our new product. The challenge is to be able to model the behavior we can observe of the legacy code so that we can enter it in the model as we do it with any other element of the environment. For doing this, the requirement document of our new product must contain some elements concerned with the legacy code. Such requirements (assumptions) have to be defined informally as we explained above.

The goal is to develop in our model the minimal interface which is compatible with the legacy code. As usual, the key is abstraction and refinement: how can we gradually introduce the legacy code in our model in such a way that we take full account of the concrete interface it offers.

Problem (2) is far more difficult than the previous one. In fact, such renovations often give very disappointing results. People tend to consider that the legacy code "is" the requirement document of the renovation. This is an error.

The first step is to write a brand new requirement document not hesitating to depart form the legacy code by defining abstract requirements which 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 to try to mimic too closely the legacy code because it might contain aspects that are not comprehensible (except for the absent legacy code programmer(s)) and that are certainly not the result of a formal modelling approach.

Our advice here is to think two times before embarking in such a light renovation. A better approach is to develop a new product. People think it might be more time and money consuming than a simple renovation: experience shows that it is rarely the case.

The Use of 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 modelling. This is an error. Set-theoretic notations are well suited to perform our system modelling and, moreover, we can understand what it means when we write a formal statement!

One can also hear very frequently that one must hide the usage of mathematical notation, because engineers will not understand them and be afraid by them: this is non-sense. Could one imagine that it is necessary to hide the mathematical notation used in the design of an electrical network because electrical engineers would be afraid by them?

Other Validation Approaches

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

These approaches are validating the solution, the software, not the problem, the global system. In each case, you construct a piece of software and then (and only then) you try to validate it (although it is not entirely the case with model checking which is also used for problem validation). For doing so, you think of a certain desired property and check that indeed your software is consistent with it. If it is not the case then you 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 than the pure development cost.

We do not think that these approaches alone are appropriate. However, we are not saying of course that one should reject them. We are just saying they might complement the modelling and proving approach.

Innovation

Big industrial corporations are often unable to innovate. They sometimes do so however provided a very large amount of money is given to them precisely for this: needless to say, it is very rare. 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 to have 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 in industry of the approach advocated in this prologue should be done through small innovative companies rather than big corporations

Education

Most 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. The basic background of a system engineer 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 be improved. Of course, it is clear that many academics will disagree with that: it is not the smallest problem one has to face. Many academics still 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 his discipline? Again, the fundamental basic discipline of a system and software engineers is (discrete) mathematics.

Two specific topics to be taught to future software engineers are (1) the writing of requirement documents (this is barely present in practical software engineering curriculum), and (2) the construction of mathematical models. Here the basic approach is a practical one: it has to be taught by means of many examples and projects to be done by the students. Experience shows that the mastering of the mathematical approach (including the proofs) is not a problem for students with a good *previous* mathematical background.

Technology Transfer

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

The incorporation in the development process of an important initial phase of requirement document writing followed by another important phase of modelling is usually regarded as dangerous as these additional phases impose some significant expenses to be done at the beginning of a project. Again here, managers do not believe that spending more initially will mean spending less at the end. However, experience shows that the expenditure is drastically reduced 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 overall initial action to be done in order to transfer a technology to industry is to perform a very significant preliminary education effort. Without that initial effort any technology transfer attempt is due to fail.

It should be noted that there exist also some fake technology transfers where people pretend using a formal approach (although they did not) just to get the "formal" stamp given to them by some authority.

Bibliography

The ideas presented in this short prologue are not new. Most of them come from the seminal ideas of Action Systems developed in the eighties and nineties. Important papers on Action Systems (among many others) are [1] and [2].

More recently, some of the ideas presented in this prologue have been put into practice. You can consult the web site [4] and, of course, read this book for more information, examples, and tool description.

References

1. R. Back and R. Kurki-Suonio. *Decentralization of process Nets with Centralized Control* 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributing Computing (1983)
2. M. Butler. *Stepwise Refinement of Communicating Systems* Science of Computer Programming (1996)
3. E. Gamma et al. *Design Patterns: Elements of Reusable Object Oriented Software* Addison-Wesley, 1995.
4. http://www.event-b.org