TECHNICAL REPORT SERIES

No. CS-TR-1151 April, 2009

Different Perspectives for Reasoning about Problems and Faults

M. Mazzara.

Abstract

This paper provides a different view for understanding prob- lems and faults with the goal of defining a method for the formal specification of systems. To accomplish this task we need to pass through a non trivial number of steps, concepts and tools where the first one, the most important, is the concept of method itself, since we realized that computer science has a proliferation of languages but very few methods. This work also proposes the idea of Layered Fault Tolerant Specification (LFTS) to make the method extensible to fault tolerant systems. The principle is layering the specification, for the sake of clarity, in (at least) two different levels, the first one for the normal behavior and the others (if more than one) for the abnormal. The abnormal behavior is described in terms of an Error Injector (EI) which represents a model of the erroneous interference coming from the environment. This structure has been inspired by the notion of idealized fault tolerant component but the combination of LFTS and EI using rely guarantee reasoning to describe their interaction can be considered one of the main contributions of this work. The progress toward this method and this way to organize fault tolerant specifications has been made experimenting on case studies presented in a dedicated section.

Bibliographical details

MAZZARA, M.

Different Perspectives for Reasoning about Problems and Faults
[By] M. Mazzara.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2009.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1150)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1150

Abstract

This paper provides a different view for understanding prob- lems and faults with the goal of defining a method for the formal specification of systems. To accomplish this task we need to pass through a non trivial number of steps, concepts and tools where the first one, the most important, is the concept of method itself, since we realized that computer science has a proliferation of languages but very few methods. This work also proposes the idea of Layered Fault Tolerant Specification (LFTS) to make the method extensible to fault tolerant systems. The principle is layering the specification, for the sake of clarity, in (at least) two different levels, the first one for the normal behavior and the others (if more than one) for the abnormal. The abnormal behavior is described in terms of an Error Injector (EI) which represents a model of the erroneous interference coming from the environment. This structure has been inspired by the notion of idealized fault tolerant component but the combination of LFTS and EI using rely guarantee reasoning to describe their interaction can be considered one of the main contributions of this work. The progress toward this method and this way to organize fault tolerant specifications has been made experimenting on case studies presented in a dedicated section.

About the author

Manuel Mazzara achieved his Masters in 2002 and his Ph.D in 2006 at the University of Bologna. His thesis was based on Formal Methods for Web Services Composition. During 2000 he was a Technical Assistant at Computer Science Laboratories (Bologna, Italy). In 2003 he worked as Software Engineer at Microsoft (Redmond, USA). In 2004 and 2005 he worked as a free lance consultant and teacher in Italy. In 2006 he was an assistant professor at the University of Bolzano (Italy) and in 2007 a researcher and project manager at the Technical University of Vienna (Austria). Between 1995 and 2007 he worked also as a system administrator, receptionist, librarian assistant and in security services. He is interested in literature, music, psychology, sport and traveling. Currently he is a Research Associate at the Newcastle University (UK) working on the DEPLOY project.

Suggested keywords

METHODS,
LAYERED FAULT TOLERANT SPECIFICATION,
PROBLEM FRAMES,
RELY/GUARANTEE

COMPUTING SCIENCE

Different Perspectives for Reasoning about Problems and Faults

M. Mazzara

TECHNICAL REPORT SERIES

No. CS-TR-1150 April, 2009

Different Perspectives for Reasoning about Problems and Faults

Manuel Mazzara

School of Computing Science, Newcastle university, UK
manuel.mazzara@newcastle.ac.uk

Abstract. This paper provides a different view for understanding problems and faults with the goal of defining a method for the formal specification of systems. To accomplish this task we need to pass through a non trivial number of steps, concepts and tools where the first one, the most important, is the concept of method itself, since we realized that computer science has a proliferation of languages but very few methods. This work also proposes the idea of Layered Fault Tolerant Specification (LFTS) to make the method extensible to fault tolerant systems. The principle is layering the specification, for the sake of clarity, in (at least) two different levels, the first one for the *normal behavior* and the others (if more than one) for the *abnormal*. The abnormal behavior is described in terms of an Error Injector (EI) which represents a model of the erroneous interference coming from the environment. This structure has been inspired by the notion of idealized fault tolerant component but the combination of LFTS and EI using rely guarantee reasoning to describe their interaction can be considered one of the main contributions of this work. The progress toward this method and this way to organize fault tolerant specifications has been made experimenting on case studies presented in a dedicated section.

1 Introduction

Dubium Sapientiae initium - Descartes

There is a long tradition of approaching Requirements Engineering (RE) by means of formal or semi-formal techniques. Although "fuzzy" human skills are involved in the process of elicitation, analysis and specification - as in any other human field - still methodology and formalisms can play an important role [22]. Anyway, the main RE problem has always been communication. A definition of communication teaches us that [11]:

> Human communication is a process during which source individuals initiate messages using conventionalized symbols, nonverbal signs, and contextual cues to express meanings by transmitting information in such a way that the receiving party constructs similar or parallel understanding or parties toward whom the messages are directed.

The first thing that we realized in building dependable software is that it is necessary to build dependable communication between parties that use different languages and vocabulary. In the above definition you can easily find the words *"similar or parallel understanding are constructed by the receiving parties"*, but for building dependable systems matching expectations (and specification) it is not enough to build a *similar or parallel understandings* since we want a more precise mapping between intentions and actions. Formal methods in system specification look to be an approachable solution. We will come back to this later when we discuss the motivations that led us to a method definition.

Object Oriented Design [8] and Component Computing [27] are just well known examples of how some rigor and discipline can improve the final quality of software artifacts besides the human communication factor. The success of languages like Java or C# could be interpreted in this sense, as natural target languages for this way of structuring thinking and design. It is also true - and it is worth reminding it - that in many cases it has been the language and the available tools on the market that forced designers to adopt object orientation principles, for example, and not vice versa. This is the clear confirmation that it is always a combination of conceptual and software tools together that create the right environment for the success of a discipline.

Semi formal notations like UML [12] helped in creating a language that can be understood by both specialists and non specialists, providing different views of the system that can be negotiated between different stakeholders with different backgrounds. The power (and thus the limitation of UML) is the absence of a formal semantics (many attempts can be found in the literature anyway) and the strong commitment on a way of reasoning and structuring problems which is clearly the one disciplined by object orientation. Many other formal/mathematical notation existed for a long time for specifying and verifying systems like process algebras (a short history by Jos Baeten in [5]) or specification languages like Z (early description in [4]) and B [2]. The Vienna Development Method (VDM) is maybe one of the first attempts to establish a Formal Method for the development of computer systems [7]. All these notations are very specific and can be understood only by specialists. The point about all these formalisms is that they are indeed notations, formal or semi-formal. Behind each of them there is a way of structuring thinking that does not offer complete freedom and thus forces designers to adhere to some discipline. But still they are not methods in the proper sense, they are indeed languages.

The goal of this paper is providing a different view for interpreting problems and faults. The overall result will be the definition of a method for the specification of systems that do not run in isolation but in the real, physical world. To accomplish this task we need to pass through a non trivial number of steps, concepts and tools where the first one, the most important, is the concept of method itself, since we realized that computer science has a proliferation of languages but very few methods. In the following we want to put more emphasis on the difference between methods and languages and, as a consequence, between formal methods and formal languages.

2 What is a method and why do we want one?

The idea of this section is defining a set of desiderata for the method we will present later in this work. We reached these ideas partly in an attempt to understand what a method basically is, and partly gaining experience and insights by experimenting with case studies. The first step will be providing definitions and examples for the notion of method and only then determining the desiderata.

2.1 What is a method: definitions and example

Firstly, we think it is important to distinguish between the words method and methodology, very similar but slightly different and often misused. We just report the Websters dictionary definitions:

> "A method is a way, technique, or process of or for doing something"

It is worth noting that the definition of method depends on the one of process:

> "a series of actions or operations conducing to an end"

The word methodology instead is defined as follows:

> Methodology is intended as the analysis of the principles of methods [...] employed by a discipline. It can be also intended as a particular procedure or set of procedures.

Thus, the word can be used to intend "a particular procedure" but the general meaning is "the analysis of the principles of methods". From now on we will use the word method to refer to a number of steps that need to be performed to reach a particular outcome, which is exactly what we want. According to these definitions when we refer to Process Algebras, for example, the words methodology and method are not correct. Anyway, it is common practice to use the word formal methods to intend formal languages in computer science. In this paper we will use the word method to intend the final result of a methodological study related to a specific context, in this case software systems specification. To properly understand what a method is and what it is not we will explore an illustrious example by Descartes, the "Discourse on the Method" (1637) [25]. It is the famous philosophical and mathematical treatise which is the source of the quotation "Je pense, donc je suis" ("I think, therefore I am"). For lack of space here it is not possible to report all the relevant parts. We are only interested in understanding how Descartes perceives a method and what is peculiar in it. Here are his four points:

1. The first was to never accept anything as true which I could not accept as obviously true; that is to say, to carefully avoid impulsiveness and prejudice, and to include nothing in my conclusions but whatever was so clearly presented to my mind that I could have no reason to doubt it.

2. The second was to divide each of the problems I was examining in as many parts as I could, as many as should be necessary to solve them.
3. The third, to develop my thoughts in order, beginning with the simplest and easiest to understand matters, in order to reach by degrees, little by little, to the most complex knowledge, assuming an orderliness among them which did not at all naturally seem to follow one from the other.
4. And the last resolution was to make my enumerations so complete and my reviews so general that I could be assured that I had not omitted anything.

It is easy to realize that a method proposes a partially ordered set of actions that need to be performed and then discharged within a specific causal relationship. The success of one action determines the following ones. Furthermore the method has to be repeatable, possibly by non experts or specialists. In figure 1 we report a graphical synthesis of the Descartes method.

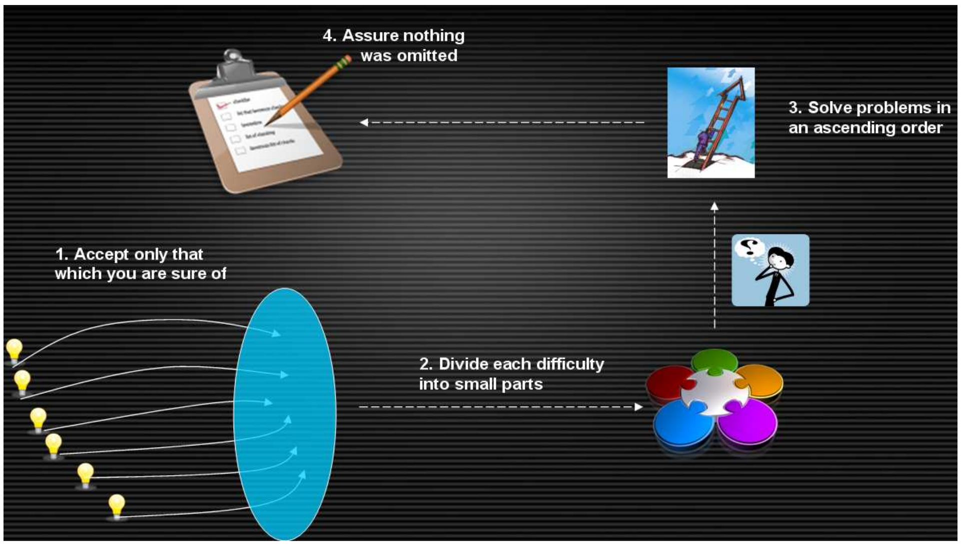
4. Assure nothing was omitted
3. Solve problems in an ascending order
1. Accept only that which you are sure of
2. Divide each difficulty into small parts

Fig. 1. The method of science

2.2 Why a method: the desiderata

The above definitions and the example are a starting point to understand better what a method is and what it is not. At this point the differences between a formal language an a formal method should be clearer. Now we have to ask ourselves why we need a method. The "why" is an interesting point, it is a meta question, a question that allows us to reason about the method looking from outside the method. The logic is what is done inside the system, in this case the formal steps performed (in some order) to reach the desired end. i.e. the method itself. The reasoning is what is done "outside the system", experimenting and seeing what happens if we change the basic rules. Reasoning "about the method" gives us a way to find out the motivations leading to a method definition. What we believe now is that the first step in building dependable software is building dependable communication between parties that have different languages/vocabulary. According to the definition of communication, Formal Methods in system specification are tools to commit on dependability since they help us in clarifying our vocabulary and providing a notation able to build a precise mapping between intentions and actions in the different stakeholders' minds. Thus a clear and precise definition of a formal method (in the actual meaning of the word) seems to be necessary at this stage. What we have understood from Descartes' lesson is that a method, and so the method we are going to propose, needs to satisfy a number of properties. The first three are taken from Descartes' method, while the last three are grabbed from our experiments with case studies which we will introduce later but we think it is worth to gather all the features together now. Our understanding of the method of science has told us that:

- It has to consist of steps to acquire knowledge
- It has to be formally defined ("phases", steps, workflow)
- It has to be repeatable (by non formal methods experts)

Then our practical experience has suggested that:

- It has to be scalable (non "ad hoc" - it has to work outside specific case studies)
- It needs abstractions (what and not how, we introduce the idea of plug-ins)
- It has to be extensible to fault tolerant behavior (we propose the idea of LFTS for this)

After having understood where we want to go and why, it is good practice to say now how we want to get there.

3 A Different Angle to See Problems

Our work in this paper focuses especially on [20] where the original idea of a formal method for the specification of systems running in the physical world originated. That paper was full of interesting ideas but still was lacking of a method in the sense we described so far. Few case studies have been analyzed according to this philosophy in [9] but still a complete method has not been reached. For this reason we think now that a more structured approach is urgent in this area. Thus, the goal of the present work is to improve our understanding of those ideas, trying to increment that contribution and to put it in an homogeneous and uniform way describing a method featuring the properties we introduced above with particular attention to fault tolerance. At the moment we have had some progress in this direction but we still need more work toward a method for the specification of fault tolerant systems. The basic idea behind [20] was to specify a system not in isolation but considering the environment in which it is going to run and deriving the final specification from a wider system where assumptions have been understood and formalized as layers of rely conditions. Here the difference between assumptions and requirements is crucial, especially when considering the proper fault tolerance aspects. We could briefly summarize this philosophy as follows:

– Not specifying the digital system in isolation
– Deriving the specification starting from a wider system in which physical phenomena are measurable
– Assumptions about the physical components can be recorded as layers of rely-conditions (starting with stronger assumptions and then weakening when faults are considered)

Sometimes we have found it useful, in the presentation of these concepts, to use the figure 2. This figure allows us to show how a computer system can be seen from a different angle, as not consisting of functions performing tasks in isolation but as relationships (interfaces/contracts) in a wider world including both the machine and the physical (measurable) reality. As we will see later this philosophy has been inspired by Michael Jackson's approach to software requirements analysis typically called Problem Frames approach [16]. The Silicon Package is the software running on the hosting machine. It should be clear that the machine itself can neither acquire information on the reality around nor modify it. The machine can only operate trough sensors and actuators. To better understand this point, we like to use a similar metaphor about humans where it is easier to realize that our brain/mind system (our Silicon Package?) cannot acquire information about the world but it can only do that through eyes, ears and so on (our sensors). In the same way it cannot modify the world if not through our arms, voice, etc (our actuators). So, as we start describing problems in the real world in terms of what we perceive and what we do (and not about our brain functioning) it makes sense to adopt a similar philosophy for computer systems consisting of sensors and actuators. Around the Silicon Package you can see a red circle representing the problem world and green small spheres representing the assumptions that need to be made regarding it. The arrows and their directions represent the fact that we want to derive the specification of the silicon package starting from the wider system. The way in which we record these assumptions is a topic for the following sections.

3.1 The method, its Steps and its Views

In this work we are structuring the method introduced in [20] according to the properties described in the previous section. To do this behind that work we recognize three macroscopic steps:

1. Define boundaries of the systems

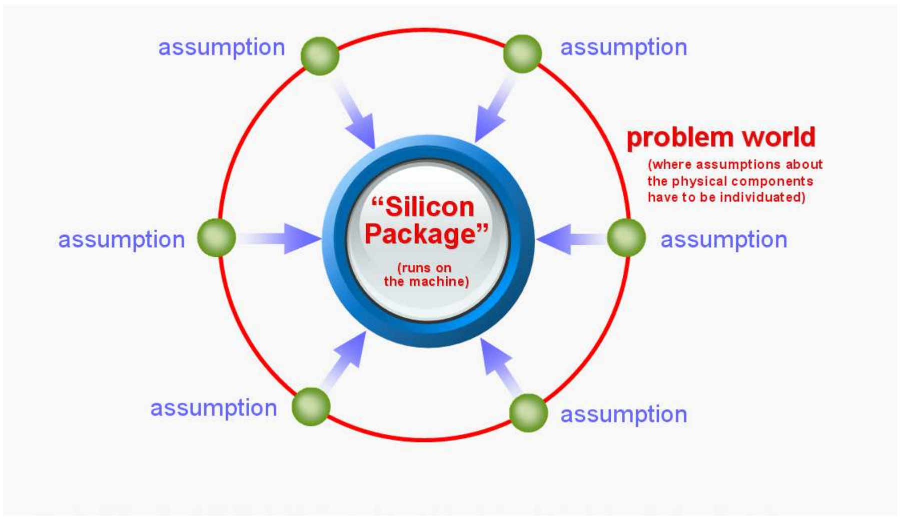

Fig. 2. Silicon Package, Problem World and Assumptions

2. Expose and record assumptions
3. Derive the specification

Our idea is to not commit to a single language/notation - we want a formal method, not a formal language - so we will define a general high level approach following these guidelines and we will suggest *reference tools* to cope with these steps. It is important noting that these are only reference tools that are *suggested* to the designers because of a wider experience regarding them from our side. A formal notation can be the final product of the method but it still needs to be not confused with the method itself. In figure 3 these steps are presented and it is showed how different tools could fit the method at different stages. We call these notations the plug-ins since they can be plugged into the different steps.

Figure 3 is a generic representation of the method where we want to emphasize the different steps that were not clearly defined previously in [20]. The reader will understand that this is still a simplification of the process. We use the word "steps" instead of "phases" since we do not want to suggest a sort of linear process which is not always applicable, in the average case (especially when coping with fault tolerance as we will discuss later). We imagine, in the general case, many iterations between the different steps. The idea of the method is to ground the view of the silicon package in the external physical world. This is the problem world where assumptions about the physical components *outside* the computer itself have to be recorded. Only after this can we derive the specification for the software that will run *inside* the computer. This more precise formalization of the method and the features the method has to exhibit is one of the main contributions of this work. The reader is probably realizing that what

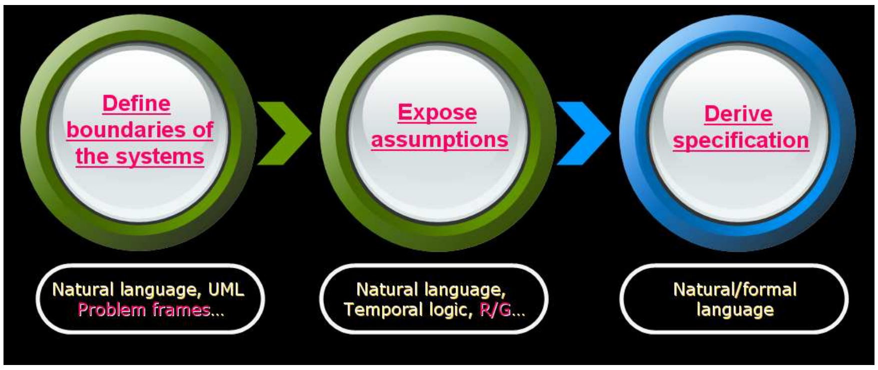
Define boundaries of the systems
Expose assumptions
Derive specification
Natural language, UML Problem frames...
Natural language, Temporal logic, R/G...
Natural/formal language

Fig. 3. Steps and Reference Tools

we are obtaining here is a method exploiting two different perspectives during the three steps.

– a *static view* defining the boundaries of the system and representing the relationships between phenomena and domains in it. Our reference tools here are Problem Diagrams [16].
– a *dynamic view* representing the interactions between different processes in the system and able to record the assumptions. Our mathematical reference tools here are rely/guarantee conditions [17, 19, 18] which regard the execution of concurrently executing (and interfering) processes.

Furthermore we need an approach to consider faulty behaviors, this will be described later in the related section. The idea behind having two different views is that different people (or stakeholders) could be possibly interested only in single aspects of the specification and be able to understand only one of the possible projections. In this way you can approach it without a full understanding of every single aspect.

3.2 Static View

Michael Jackson is well known for having pioneered, in the seventies (with Jean-Dominique Warnier and Ken Orr) the technique for structuring programming basing on correspondences between data stream structure and program structure [14]. Jackson's ideas acquired then the acronym JSP (Jackson Structured Programming). In his following contribution [15] Jackson extended the scope to systems. Jackson System Development (JSD) already contained some of the ideas that made object-oriented program design famous. Figure 4 shows Jackson's main contribution to the field. In this section we describe our reference tool for representing the relationships between phenomena and domains of the system we want to specify using Problem Diagrams [16]. Context Diagrams and Problem Diagrams are the graphical notations introduced by Michael Jackson (in the time frame 1995/2001) in his Problem Frames (PF) approach to software requirements analysis. This approach consists of a set of concepts for gathering requirements and creating specifications of software systems. As previously explained in this work the new philosophy behind PF is that user requirements are here seen as being about relationships in the operational context and not about functions that the software system must perform. It is someway a change of perspective with respect to other requirements analysis techniques.

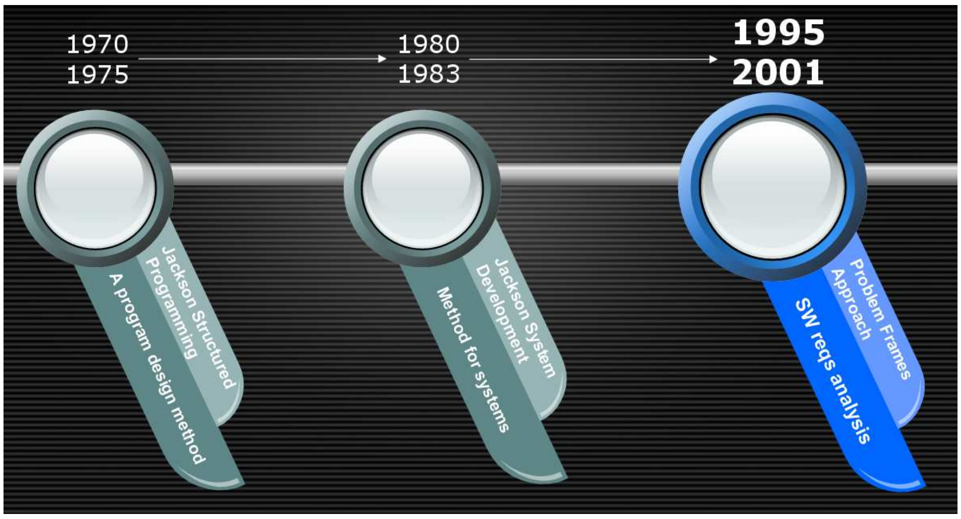

Fig. 4. Jackson Methods

The entire PF software specification goal is modifying the world (the problem environment) through the creation of a dedicated machine which will be then put into operation in this world. The machine will then operate bringing the desired effects. The overall philosophy is that the problem is located in the world and the solution in the machine. The most important difference with respect to other requirements methodologies is the emphasis on describing the environment and not the machine or its interfaces. Consider, for example, the Use Case approach [6]. Here what we do is specify the interface and the focus is on the interaction user/machine. With PF we are pushing our attention beyond the machine interface, we are looking into the real world. The problem is there and it is worth to start there. The first two points of the ideas taken from [20] (not specifying the digital system in isolation and deriving the specification starting from a wider system in which physical phenomena are measurable) can be indeed tracked back, with some further evolution, to [16]. We are using PF to develop a method for specification of systems, i.e. a description of the machine behavior. But, before doing that, we start understanding the problem.

Context Diagrams The modeling activity of a system should start using this kind of diagram in the PF philosophy. By means of it we are able to identify the boundaries of the system, where a system is intended as the machine to be designed (software + hardware) and its domains with their connections (in terms of shared phenomena). It is part of what we call a static view of the system.

Context Diagrams contain an explicit and graphical representation of:

– the machine to be built
– the problem domains that are relevant to the problem
– the interface (where the Machine and the application domain interact)

A domain here is considered to be a part of the world we are interested in (phenomena, people, events). A domain interface is where domains communicate. It does not represent data flow or messages but shared phenomena (existing in both domains). Figure 5 shows a simple scenario. The lines represent domain interfaces, i.e. where domains overlap and share phenomena.

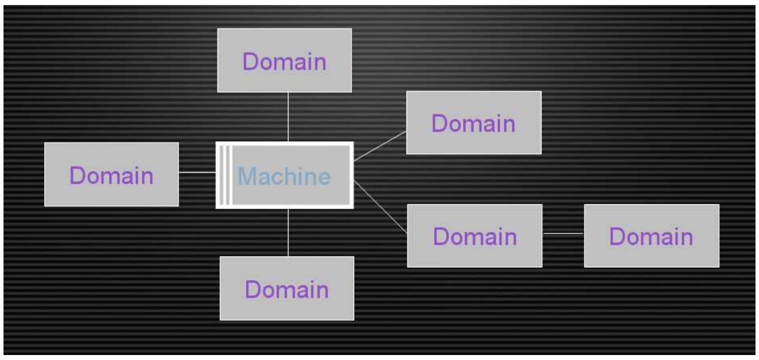

Fig. 5. Contex Diagram

Problem Diagrams The basic tool for describing a problem is a Problem Diagram which can be considered a refinement of a Context Diagrams. This should be the 2nd step of the modeling process. A problem diagram shows the requirements on the system, its domains, and their connections. It is still part of a static view of the system but better represents the assumptions about the system and its environment. They are basic tools to describe problems. To the information contained in context diagrams they add:

– dotted oval for requirements
– dotted lines for requirements references

Figure 6 shows a scenario where the Silicon Package is in charge of monitoring the patients conditions. We believe that the first step of the specification method (define boundaries of the systems) can be accomplished by means of this tools. Thus we use Problem Diagrams as a reference tool for our research but still, as said, not constraining it to a specific notation or language.

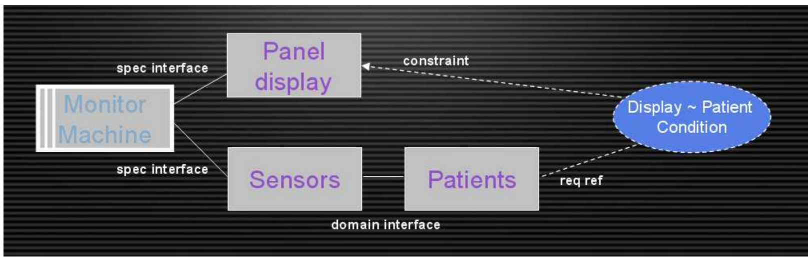

Fig. 6. Problem Diagram

3.3 Dynamic View

Problem Diagrams taken from the PF approach are a notation that forces us to think about the problem in the physical world instead of focusing immediately on the solution. We believe that they represent an effective tool to define the precise boundaries of the specification we are working on. Summarizing they represent:

1. the machine
2. the problem domains
3. the domain interfaces
4. the requirements to bring about certain effects in the problem domains
5. references in the requirements to phenomena in the problem domains

Once the domains of the context we are working on, their phenomena and the relative overlap have been understood, it will be necessary to focus on the "border" between the Silicon Package and the real world. It is necessary to distinguish between assumptions and requirements and we need a tool to record the assumptions. Our system will be composed of interacting parts and each of these parts will interact with the world. The world itself has to be understood in term of assumptions about normal/abnormal behavior and a model of fault need to be considered. For all these reason we introduce the concept of *dynamic view* which represents the interactions between processes in the system and between the system and the world. To record our assumption (as we will see layers of assumption for fault tolerance) we use a mathematical reference tool, i.e. rely/guarantee conditions [17, 19, 18] which regard the execution of concurrently executing processes. R/G conditions are a powerful abstraction for reasoning about interference originated in the Hoare logic idea of preconditions and postcondition [13]. The purpose is providing a set of logical rules to reason about the correctness programs. We will explain the idea through examples, for more details please consider the literature in this regard. As the reader will realize in this section, rely conditions can be used to record assumptions in the overall context of the proposed method. But, as stated in [23], when they show too much complication that might be a warning indicating a messy interface.

Preconditions and Postcondition To understand the power of the R/G reasoning it is necessary to realize how preconditions and postconditions can help in specifying a software program when interference does not play its role. What we have to describe (by means of logical formulas) when following this approach is:

1. the input domain and the output range of the program
2. the precondition, i.e. the predicate that we expect to be true at the beginning of the execution
3. the postcondition, i.e. the predicate that will be true at the end of the execution provided that the precondition holds

Preconditions and postconditions represent a sort of contracts between parties: provided that you (the environment, the user, another system) can ensure the validity of a certain condition, the implementation will surely modify the state in such a way that another known condition holds. There is no probability here, it is just logic: if this holds that will hold. And the input-output relation is regulated by a predicate that any implementation has to satisfy.

We show the example of a very simple program, the specification of which in the natural language may be:

Find the smallest element in a set of natural numbers

This very simple natural language sentence tells us that the smallest element has to be found in *a set of natural numbers*. So the output of our program has necessarily to be a natural number. The input domain and the output range of the program are then easy to describe:

$$I/O : \mathcal{P}(\mathbf{N}) \rightarrow \mathbf{N}$$

Now, you expect your input to be a set of natural numbers, but to be able to compute the min such a set has to be non empty since the min is not defined for empty sets. So the preconditions that has to hold will be:

$$P(S) : S \neq \emptyset$$

Provided that the input is a set of natural numbers *and* it is not empty, the implementation will be able to compute the min element which is the one satisfying the following:

$$Q(S, r) : r \in S \wedge (\forall e \in S)(r \leq e)$$

Given this set of rules, the input-output relation is given by the following predicate that needs to be satisfied by any implementation f:

$$\forall S \in \mathcal{P}(\mathbf{N})(P(S) \Rightarrow f(S) \in \mathbf{N} \wedge Q(S, f(S)))$$

Interference The example just showed summarizes the power (and the limitations) of these kinds of abstractions. To better understand the limitations consider figure 7 where interference trough global state is depicted. The two processes alternate their execution and access to the state. The global state can consist of shared variables or can be a queue of messages if message passing is the paradigm adopted (at the end the two paradigms are equivalent). This figure shows exactly the situations described in [19], quoting precisely that work:

> As soon as the possibility of other programs (processes) running in parallel is admitted, there is a danger of "interference." Of more interest are the places where it is required to permit parallel processes to cooperate by changing and referencing the same variables. It is then necessary to show that the interference assumptions of the parallel processes coexist.

Another quote from [10] says:

> The essence of concurrency is interference: shared-variable programs must be designed so as to tolerate state changes; communication-based concurrency shifts the interference to that from messages. One possible way of specifying interference is to use rely/guarantee-conditions.

In the case in which we consider interfering processes we need to accept that the environment can alter the global state but the idea behind R/G is that we impose these changes to be constrained. Any state change made by the environment (other concurrent processes with respect to the one we are considering) can be assumed to satisfy a condition called R (rely) and the process under analysis can change its state only in such a way that observations by other processes will consist of pairs of states satisfying a condition G (guarantee). Thus, the process *relying* on the fact that a given condition holds can *guarantee* another specific condition. An example is needed.

Greatest Common Divisor Consider the two following simple pieces of code, the cooperation of which calculates the Greatest Common Divisor:

```
P1:
while(a<>b){
if(a > b)
   a := a-b;
}
```

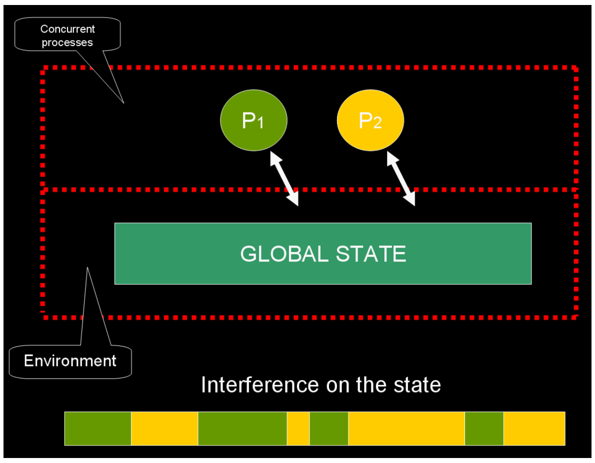

Fig. 7. Interference trough global state

```
P2:
while(a<>b){
  if(b > a)
    b := b-a;
}
```

P1 is in charge of decrementing a and P2 of decrementing b. When $a = b$ will evaluate to true it means that one is the Greatest Common Divisor for a and b. The specification of the interactions is as follows:

$$R_1 : (a = \overline{a}) \wedge (a \geq b \Rightarrow b = \overline{b}) \wedge (GCD(a, b) = GCD(\overline{a}, \overline{b}))$$
$$G_1 : (b = \overline{b}) \wedge (a \leq b \Rightarrow a = \overline{a}) \wedge (GCD(a, b) = GCD(\overline{a}, \overline{b}))$$
$$R_2 = G_1$$
$$G_2 = R_1$$

Here the values $\overline{a}$ and $\overline{b}$ are used instead of a and b when we want to distinguish between the values before the execution and the values after. P1 relies on the fact that P2 is not changing the value of a and $a \geq b$ means no decrements for b have been performed. Furthermore the GCD did not change. Specular situation is for the guarantee condition. Obviously, what is a guarantee for P1 becomes a rely for P2 and vice versa. Figure 8 is a graphical representation of the GCD example showing that P1 is only reading b but updating a, the opposite for P2.

3.4 Need for Extension (of Jackson's Diagrams)?

The objective of a PF analysis is the decomposition of a problem into a set of subproblems, where each of these matches a problem frame. A problem frame is a problem pattern, i.e the description of a simple and generic problem for

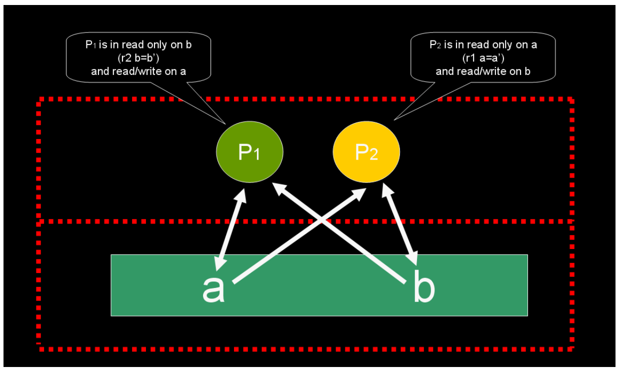
P1 is in read only on b (r2 b=b') and read/write on a
P2 is in read only on a (r1 a=a') and read/write on b
P1
P2
a
b

Fig. 8. GCD example

which the solution is already known. There are four main patterns plus some variations:

– required behavior (the behavior of a part of the physical world has to be controlled)
– commanded behavior (the behavior of a part of the physical world has to be controlled in accordance with commands issued by an operator)
– information display (a part of the physical world states and behavior is continuously needed)
– simple workpieces (a tool is needed for a user to create/edit a class of text or graphic objects so that they can be copied, printed...)

Our perception is that, when describing the behavior of interfering processes - especially when faults are considered as a special case of interference (see next section) - the diagrams and the patterns provided are not powerful enough. We need further refinement steps filling the gap between the static and the dynamic view to complete the specification process. Now we briefly describe these ideas that needs further work and can be considered an open issue.

Interface Diagram In a 3rd step of the modeling process, we want to represent an external, static view of the system. We need a further refinement of the Problem Diagram able to identify the operations of the system and its domains, and the input/output data of these operations (with their types). The relationship of these with the requirements identified in the Problem Diagram has to be represented at this stage.

Process Diagram In a 4th step of the modeling process, the whole system is represented as a sequential process and each of its domains as a sequential process. Concurrency within the system or within its domains is modeled by representing these as two or more subcomponents plus their rely and guarantee conditions. This is an external, dynamic view of the system and its domains.

4 A Different Angle to See Faults

Testing can never guarantee that software is correct. Nevertheless, for specific software features - especially the ones involving human actions and interactions - rigorous testing still remains the best choice to build the desired software. We know very little about human behavior, there are few works trying to categorize, for example, human errors in such a way that we can design system that can prevent bad consequences [26] but this goes far beyond the scope of this work. Here we want to focus on the goal of deploying highly reliable software in terms of aspects that can be quantified (measured), for example the functional input/output relation (or input/output plus interference, as we have seen). In this case formal methods and languages could provide some support. The previous sections discussed how to derive a specification of a system looking at the physical world in which it is going to run. No mention has been made of fault tolerance and abnormal situations which deviate from the basic specification. The first thing the reader will realize is that the method we defined does not cope with these issues but it does not prevent fault tolerance from playing a role. The three steps simply represent what you have to follow to specify a system and they do not depend on what you are actually specifying. This allows us to introduce more considerations and to apply the idea to a wider class of systems. Usually in the formal specification of sequential programs widening the precondition leads to make a system more robust. The same can be done weakening rely conditions. For example, if eliminating a precondition the system can still satisfy the requirements this means we are in presence of a more robust system. Here we will follow this approach presenting the notion of Layered Fault Tolerant Specification (LFTS) and examining the idea of fault as interference [10], i.e. a different angle to perceive system faults. Quoting [10]:

> The essence of this section is to argue that faults can be viewed as interference in the same way that concurrent processes bring about changes beyond the control of the process whose specification and design are being considered.

Here we are introducing the idea of Layered Fault Tolerant Specification (LFTS) combining it with the approach quoted above making use of rely/guarantee reasoning. The principle is layering the specification, for the sake of clarity, in (at least) two different levels, the first one for the *normal behavior* and the others (if more than one) for the *abnormal*. This approach originated from the notion of idealized fault tolerant component [21] but the combination of LFTS and rely guarantee reasoning can be considered one of the main contributions of this work.

4.1 Fault Model

First, when specifying concurrent (interfering) processes, we need to define which kind of abnormal situations we are considering. We basically need to define a

Fault Model, i.e.what can go wrong and what cannot. Our specification will then take into account that the software will run in an environment when specific things can behave in an "abnormal" way. There are three main abnormal situations in which we can incur, they can be considered in both the shared variables and message passing paradigm:

– Deleting state update: lost messages
– Duplicating state update: duplicated messages
– Additional state update (malicious): fake messages created

The first one means that a message (or the update of a shared variable) has been lost, i.e. its effect will not be taken into account as if it never happened. The second one regards a situation in which a message has been intentionally sent once (or a variable update has been done once) but the actual result is that it has been sent (or performed) twice because of a faulty interference. The last case is the malicious one, i.e. it has to be done intentionally (by a human, it cannot happen only because of hardware, middleware or software malfunctioning). In this case a fake message (or update) is created from scratch containing unwanted information.

Our model of fault is represented by a so called *Error Injector* (EI). The way in which we use the word here is different with respect to other literature where Fault Injector or similar are discussed. Here we only mean a model of the erroneous behavior of the environment. This behavior will be limited depending on the number of abnormal cases we intend to consider and the EI will always play its role respecting the RG rules we will provide. In the example we will show in the following we are only considering the first of the three cases, i.e the Fault Injector is only operating through lost messages.

A contribution of this work is the organization of the specification in terms of layers of Rely/Guarantee conditions. In order to do this we introduce the idea of EI as a model of the environment and we need to describe how the EI will behave and how we can limit it. Here a process will rely on a specific faulty behavior and, given that, will guarantee the ability to handle these situations. More in detail:

– Rely: the Error Injector (environment) interferes with the process (changing the global state) respecting his G (superset of the programs R) for example, only lost messages can be handled (next example)
– Guarantee: The process provided this kind of (restricted) interference is able to handle exceptional/abnormal (low frequency) situations

All the possibilities of faults in the system are described in these terms and the specification is organized according to the LFTS principle we are going to describe.

4.2 LFTS: how to organize a clear specification

The main motto for LFTS is: "Do not put all in the normal mode". From the expressiveness point of view, a monolithic specification can include all the aspects, faulty and non faulty of a system in the same way as it is not necessary to organize a program in functions, procedures or classes. The matter here is pragmatics, we believe that following the LFTS principles a specification can be more understandable for all the stakeholders involved.

The specification has to be separated in (at least) two layers, one for the *Normal Mode* and one (or more) for the *Abnormal Mode*. More specifically:

– Normal mode: an operation usually runs in normal mode respecting his interface with the world determined by P/Q
– Fault interference: in low frequency cases the abnormal mode is activated (exception handler, forward recovery)

Figure 9 shows the organization of a process (dashed rectangles) in a main part and a *recovery handler* part where both interact through the global state with other processes and the Error Injector (represented by a devil here).

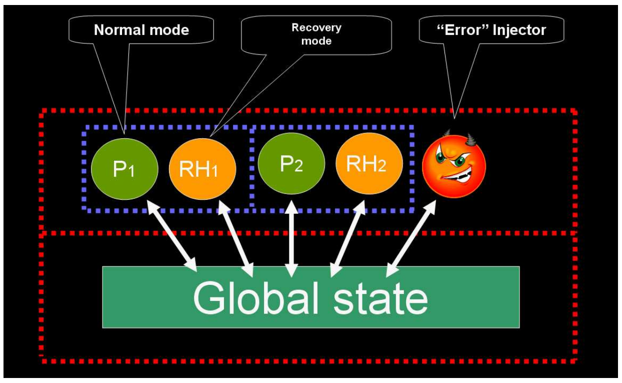

Fig. 9. Error Injector

It is worth noting the limitations of this way of operating. Self error detection and self recovery cannot be addressed by this model since EI is a representation of the environment external to the process itself. So faulty behavior due to internal malfunctioning is not what we want to represent here.

4.3 Example of Specification of Interference

For a better understanding of how we can exploit this idea of treating faults as extraordinary interference with a low frequency but still manageable with the same tools used for normal behavior we introduce a very simple example. First we consider an even simpler example without interference, then we introduce interference to investigate the differences and how we cope with it.

Increments without Interference Let us consider the following piece of code:

```
C(n):
n' := n;
while (n'>0){
      n' := n-1;
      count ++
}
return count;
```

C is a very simple program which decrements its input while reaching zero. While decrementing the input it increments a counter with the effect that, at the end, the counter will obviously reach the original value of the input. The specification of C in terms of Pre and Post conditions is given as follows:

$$I/O : \mathbf{N} \rightarrow \mathbf{N}$$

The input (n) and the output (*count*) are natural numbers. The precondition that has to hold is:

$$P(count) : count = 0$$

since we expect the counter to be zero at the beginning. Provided that the input is a natural number *and* the counter is zero, the execution will satisfy the following:

$$Q(n, count) : count = n \wedge \overline{n} = 0$$

Without any interference, the specification of C only requires that the input-output relation satisfy the predicate:

$$\forall a \in \mathbf{N}(P(a) \Rightarrow C(a) \in \mathbf{N} \wedge Q(a, C(a)))$$

Increments with Faulty Interference Let us consider the same piece of code:

```
C(n):
n':= n;
while (n'>0){
      n' := n'-1;
      count ++
}
return count;
```

but running in an environment where the following EI is also running:

```
EI(n'):
if (n'>0){
    n' := n'+1;
}
```

The role of this EI here is to model the deletion of state updates as in the first of the three cases discussed above. The specification of C as expressed so far is too simple to be able to manage this kind of situations. Even if we are not handling malicious updates, the basic formulation we provided so far needs to be properly incremented because without any changes the desired implication cannot be satisfied:

$$\forall a \in \mathbf{N}(P(a) \not\Rightarrow C(a) \in \mathbf{N} \wedge Q(a, C(a)))$$

What we have to do is restructure the implementation and to pass from pre and post conditions to rely/guarantee in the specification. Let us consider the following modification:

```
C(n):
n':= n;
while (n'>0){
   if n'+ count = n then {
    n' := n'-1;
    count ++
 } else {
    n' := n-count-1
}
}
return count;
```

As the reader will understand what we have done is simply add a recovery handler and a recovery mode based on the evaluation of the condition $n{+}count = n$ which is able to flag the presence of an unwanted interference (a deletion of an increment). The recovery block is able to cope with abnormal situations provided that faults are restricted in behavior (and that it is known in advance). Thus, provided that a restricted interference happens the program is still able to satisfy the post condition (and the specification). The normal mode here is the simple code:

```
n' := n'-1;
count ++
```

while the recovery handler is

```
n' := n-count-1
```

and, as represented in figure 9, C is running in an environment which is shared with EI. The specification we want in this case is different from the previous one and it is expressed, in terms of R/G conditions, as follows:

$R_C : (\overline{n} = n) \wedge (\overline{count} = count) \wedge (\overline{n'} > n')$
$G_C : n' = n - count - 1$
$R_I = true$

$G_I = n' > \overline{n'}$

It is worth noting that there is no rely condition (to be precise there is one always true) for the Error Injector, indeed it would not be reasonable to expect that the processes we are specifying would behave in a way so as to satisfy the needs of a fault model. Instead, EI is guaranteeing that it will only increment n' - it is the case of having only state update deletion (an increment deletes a decrement) as pointed out previously. Decided the EI behavior limitation (and thus decided the fault model) we can design our specification. From the EI specification C can rely on the fact that n and *count* will be never modified while *n'* will be only modified in a specific way (incremented). Now, with the addition of a layer in the program and in the specification we are still able to guarantee an (extended) desired behavior by means of the G_C condition which says that *n'* will always be consistent with the value of *count* preserving the invariant $n' = n - count - 1$, i.e. the summation of *n'* and *count* will always be equal to $n - 1$. This will ensure that the postcondition $count = n \wedge \overline{n} = 0$ will hold at the end like in the case without interference. This simple example shows how the LFTS principles can provide a clear specification (with respect to a monolithic one) ensuring, at the same time, that a desired postcondition holds.

5 How did we get to these ideas?

So far we have used many small toy examples to show our ideas and the principles we were presenting. Actually, we have got some progress toward a method and a way to organize specifications not only by means of these examples (that have been developed later for dissemination purposes) but experimenting on more complex case studies. In this section, we will show how they brought us to a better definition of the method and to the idea of LFTS.

5.1 The Transportation Example

Here we consider a case study taken from [3], the train system, where the goal was showing the power of modeling and formal reasoning by means of Event-B examples. We chose this scenario since we believe it is particularly realistic (it has been developed after some work with real train systems) and still manageable (with a limited set of initial requirements: 39). This case study taught us how to distinguish between assumptions and requirements and helped us in finding a better structure for the method initially presented in [20]. We will show here how this example can be approached with the three steps method. The first thing to do is deciding the bounds of specification (step 1). We will show how the boundaries can be broadened to include the external world. In the second step we will discuss how to separate assumptions and requirements, how to expose and record assumptions and how different sets of requirements and assumptions will imply a different specification and then implementation. In the third step we will assume the existence of an already designed network infrastructure (with sensors etc...) to show a specific example of implementation. At the end we will show how to make use of rely conditions for this specific implementation.

Requirements Taxonomy The Train System requirements are organized as follows:

- Environment: concerned with the structure of the track networks and its components
- Functional: dealing with the main functions of the system
- Safety: ensuring that non classical accidents could happen
- Movement: ensure that a large number of trains may cross the network at the same time
- Train: define the implicit assumptions about the behavior of trains
- Failure: define the various failures against which the system is able to react without incidents

What it is important to realize here is the way in which the interference over a global state is considered using the approach showed in the small GCD example. In the following, the specification will be showed, after a discussion about the way in which it has been obtained, and the interaction between the different operations will be constrained in a similar way but in a system with potentially higher level of concurrency.

Step 1: defining the boundaries of the system This example is about how to clarify the requirements in the real world before trying to specify the software which sits within the system. This process naturally identifies assumptions about the physical components which can be recorded as rely-conditions. One of the main principles of this approach is not specifying a system in isolation but starting moving the system boundaries outwards (what is called ”pushing out the boundaries of the system” in [20]). What is the wider system in which physical phenomena are measurable? What is the actual general purpose of the Train System? It is allowing trains to move safely from a place X to a place Y. How does this help us in identifying system requirements? We can recognize that the FUN-1 requirement of the system specification [3] expresses basically this need:

The goal of the train system is to
safely control trains moving on a track network

If we move the boundary outwards further we can say that the purpose of the system is allowing people to reach their destination safely. Considering this we could split FUN-1 in two properties (without referring to any specific implementation yet):

1. Safety property (nothing bad can happen)
2. Liveness property (something good has to happen)

We can express these two properties more in detail as follows:

Safety: *Trains will never collide*

Liveness: *Trains will move from their origin to their destination*

Req FUN-1 is general enough to allow this separation, anyway we are interested in modeling only the safety property delegating liveness to a scheduler or, theoretically to a manual management performed by operators/engineers. For the sake of simplicity, the specification will start with this requirement only. All the other requirements presented in [3] refer to concepts like blocks, routes and signals that can describe either a set of assumptions about the environment or a specific implementation of FUN-1. We will say more about this later.

Step 2: exposing and recording assumptions Now it is crucial to discriminate between *requirements for the system* and *assumptions about the real world.* In this example it was important to ask if we are in charge of designing the whole railway/track with sensors, signals, etc or not. If not, many requirements in the ENV group (and the given block structure) can be considered as assumptions taken from the already existing environment. Otherwise, all these can be seen as requirements but referring to a specific implementation and should not be introduced now but only later, in the last step. For example, the requirement ENV-13:

A signal can be red or green. Trains
are supposed to stop at red signals

is an example of mixing requirements and assumptions in the same statement. So determining the assumption (and being able to separate them from requirements) is the main goal of this step. In this example we suppose to be the designer of the whole track and we want trains to move from city X to city Y. There are many possible implementations for the presented requirements, we will look into the details of only one (which is the one given in [3]). Before looking at that it is easy to understand that the simplest possible implementation is the one requiring that no train can cross the network. This is an implementation where the Safety property is preserved (but Liveness is not). Although we are interested mainly in this property here, a better thing to do would be allowing only one train on the track between X and Y. This means basically that the rail connecting two cities will be reserved for a single train. Obviously this implementation respects both the safety and liveness requirements described above. But it is also easy to realize that it is simply unfeasible because of the very low efficiency, very low exploitation of the available resources and because of the expensiveness (time and money).

A more reasonable implementation is actually the one that in [3] is simply used for the modeling purposes. The scope here is different from what has been done there, for this reason we did not assume this implementation as given but we wanted to go through the entire discussion. The point was learning the lesson about determining wider boundaries including the external environment and distinguishing between requirements and assumptions. So we analyzed the entire process carefully. Now we are ready to present this implementation. Figure 10 represents an example of the infrastructure. It is made of:

1. Blocks: a track is made of a number of fixed blocks as showed in figure 10.
2. Routes: blocks are always structured in a number of statically pre-defined routes. Each route represents a possible path that a train may follow. Routes define the various ways a train can cross the network. A route is composed of a number of adjacent blocks forming an ordered sequence. For example a route in figure 11 is LABDKJN.
3. Points: a track contains these special components. A point may have two positions: directed or diverted. These components are attached to a given block. And a block contains at most one special component. In figure 10, B and D, for example, they both contain points. In figure 11, point B is directed while point D is diverted.
4. Signals: each route is protected by a signal (Red/Green). It is situated just before the first block of each route and it must be clearly visible by train drivers. When a signal is red the corresponding route cannot be used by an incoming train.

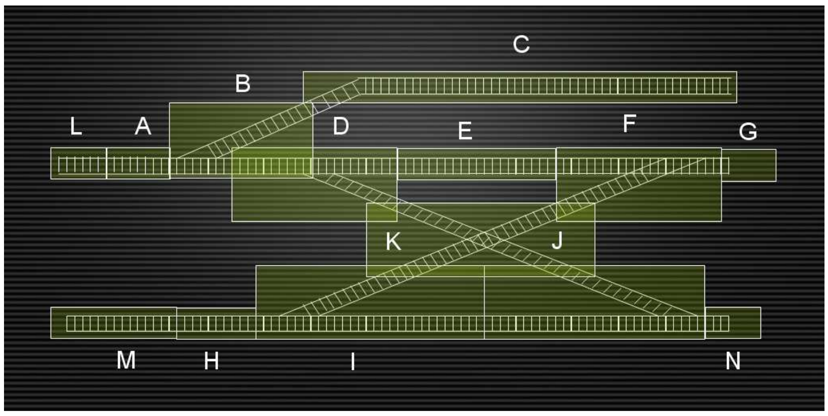

Fig. 10. The network infrastructure

The idea is to have each block of a reserved route freed as soon as the train does not occupy it anymore. It is not scope of this work to describe entirely the case study but only to describe the insights we have gained in the process of working toward a method. The reader who can find it hard to abstract over few details should refer to [3] for the detailed description of this scenario. In the next section we will focus on the reserving routes system, i.e. the process of reserving

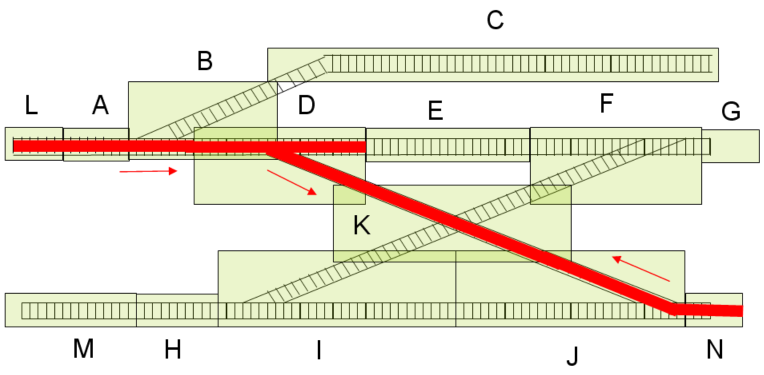

Fig. 11. Example of a route

a route on a train request, freeing it and letting the train occupy block by block freeing each block when passed.

We have also decided to abstract over concepts like time or distances. In a real system these two concepts would play a significant role in making the trip safe. Here we have decided for an infrastructure composed by blocks and we will never have two trains in the same block to avoid collisions. In a real system, for example, usually this can happen for trains heading to the same direction: in this case it will be the driver's responsibility to stop the train properly to avoid the collision. So, actually, the constraint of not having two trains in the same block would only apply in the case they are moving in opposite directions. Here we are simplifying this aspect but we are not relaxing the safety requirement anyway, since the block constraint we are proposing is strong enough to guarantee that there will be no train collision. We will see the infrastructure as a set of routes from which you cannot really infer which one is adjacent to the other. This is because we are focusing on safety purposes. We indeed do not cope with liveness, we assume that these kind of problems are managed by a scheduler which is another system already running. A graph structure would be probably more adequate in case we want to focus on the scheduler having Liveness coming into play. This abstraction simplifies our work without being in contradiction with the original philosophy of grounding the system in the physical world. We have only decided that the border between the system and the real world here will consist of the sensors and actuators necessary to make such an infrastructure working.

Step 3: deriving the formal specification Now we define the basic machinery for the formal specification. We need four finite sets for the purpose:

- `T`, a finite set of trains (t a variable ranging over it)
- `B`, a finite set of blocks (b a variable ranging over it)

– R, a finite set of routes (r a variable ranging over it)
– P, a finite set of points (p a variable ranging over it)

The safety requirement will be modeled as a total function mapping blocks to trains: $\mathtt{B} \rightarrow \mathtt{T}$ (`train`). This is how we impose to have a single train on a block. To avoid collisions by trains we also need a way to associate trains to routes, once the train has reserved a specific route. We use the function: $\mathtt{T} \rightarrow \mathtt{R}$ (`route`). A route is then composed by blocks, at least one: $\mathtt{R} \rightarrow \mathtt{B}^+$ (`blocks`) and in a route a block has a next element: $\mathtt{B} \rightarrow \mathtt{B}$ (`next`). Blocks can be free or occupied : $\mathtt{B} \rightarrow \{free, occupied\}$ (`status`) and are associated to points: $\mathtt{B} \rightarrow \mathtt{P}$ (`point`) that can be oriented in two different ways: $\mathtt{P} \rightarrow \{directed, diverted\}$ (`direction`). Routes can be available or reserved: $\mathtt{R} \rightarrow \{available, reserved\}$ (`availability`) and each route is associated with a predefined points orientation: $\mathtt{R} \rightarrow (\mathtt{P} \rightarrow \{directed, diverted\})$ (`orientation`). We rely on the fact that the sensors with which a block is equipped can always detect the presence of a train (for $\mathtt{B} \rightarrow \mathtt{T}$). We assume that if we want to reserve a point, it will be promptly positioned. We do not model these "low level" aspects here (for $\mathtt{T} \rightarrow \mathtt{R}$). We rely also on the fact that each route has a first block: $\mathtt{R} \rightarrow \mathtt{B}$ (**first**), a last block: $\mathtt{R} \rightarrow \mathtt{B}$(**last**), and they are different: $first(R) \neq last(R)$.

The mathematical machinery defined so far can be considered part of the global state on which the five operations we are going to define operate: they are related to the process of route reservation and freeing plus the entrance, proceeding and exit of a train to and from a route. These are the operations concerned with the specification of our safety requirement. Liveness is not discussed, we only move a train from one end of a route to the other without investigation about the way in which the routes are previously organized. For each operation the notation below indicates the data needed and what we expect from that data plus the way in which the global state will be modified.

Operation	$RouteReserving\ (t : \mathtt{T}, r : \mathtt{R})$
Rely	$availability(r) = available$
	$\forall\, b \in blocks(r)\ (status(b) = free)$
Guarantee	$availability(r) := reserved$
	$\forall\, b \in blocks(r)\ (status(b) := occupied)$
	$route(t) := r$
	$\forall\, p \in P\ (direction(p) := orientation(r)(p))$

Given a train and a route, this operation guarantees three mappings to be properly updated, provided that the given route is available and the related blocks are free. The three mappings are first the one between points and directions, second the one between trains and routes (as a record of the overall track status) and last the association between blocks and their occupancy status. These represent the part of the global state of interest for this operation.

Operation	$RouteFreeing\ (t : \mathtt{T})$
Rely	$\forall b \in blocks(route(t))\ (status(b) = free)$
Guarantee	$availability(route(t)) := available$
	$route(t) := null$

Given a train the related route is identified. The effect on the state is a modification of the mapping where the train is associated to the null route and, provided that all the blocks in the route are free, the route itself can be freed. This operation has a simpler definition with respect to the reservation because the blocks are freed by the *ExitRoute* while the points direction does not need to be modified when freeing a route.

Operation	*EnterRoute* $(t : \mathtt{T})$
Rely	$\mathit{availability}(\mathit{route}(t)) = \mathit{reserved}$
	$\wedge\ \mathit{status}(\mathit{first}(\mathit{route}(t))) = \mathit{free}$
Guarantee	$\mathit{status}(\mathit{first}(\mathit{route}(t))) := \mathit{occupied}$

This operation corresponds to a train entering the first block of a route. The first block must be unoccupied before the operation and it will be occupied afterward. It can be accessed only by trains that have already reserved a route.

Operation	*MovingOnRoute* $(t : \mathtt{T}, b : \mathtt{B})$
Rely	$\mathit{availability}(\mathit{route}(t)) = \mathit{reserved}$
	$\wedge\ b \in \mathit{blocks}(\mathit{route}(t))$
	$\wedge\ \mathit{status}(\mathit{next}(b)) = \mathit{free}$
Guarantee	$\mathit{status}(b) := \mathit{free}$
	$\mathit{status}(\mathit{next}(b)) := \mathit{occupied}$

This operation corresponds to the occupancy of a block which is different from the first block of a reserved route. It can be accessed only by trains that have already reserved a route. The current block has to belong to the route and the next one can be occupied only when it is free. The occupation of the next block implies that the current one becomes free.

Operation	*ExitRoute* $(t : \mathtt{T}, b : \mathtt{B})$
Rely	$\mathit{availability}(\mathit{route}(t)) = \mathit{reserved}$
	$\wedge\ b \in \mathit{blocks}(\mathit{route}(t))$
	$\wedge\ \mathit{next}(b) = \emptyset$
Guarantee	$\forall b \in \mathit{blocks}(\mathit{route}(t))\ \mathit{status}(b) := \mathit{free}$

This operation corresponds to the train exit out of the route. It can be accessed only by trains that have already reserved a route and it is responsible to free all the blocks in that route.

LFTS for the Train System Here we consider the Train System in a less ideal world than the one analyzed before. In this world, the EI plays its role, for the sake of simplicity, changing the global state only according to the "lost messages" condition. The global state of the system needs to be modified for the EI to implement its changes. Now, in the network, sensors and actuators can actually fail and some state update could be not performed. Thus, let us modify the `availability` function in such a way as to include a third option: $\mathtt{R} \rightarrow \{\mathit{available}, \mathit{reserved}, \mathit{maintenance!}\}$ (`availability`). The *RouteReserving* operation can be extended as follows:

Operation	$RouteReserving\ (t : \mathtt{T}, r : \mathtt{R})$
Rely	$availability(r) = available$
	$\forall\ b\ \in\ blocks(r)\ (status(b) = free)$
Rely $\approx$	$availability(r) = available$
Guarantee	$availability(r) := reserved$
	$\forall\ b \in\ blocks(r)\ (status(b) := occupied)$
	$route(t) := r$
	$\forall\ p \in\ P\ (direction(p) := orientation(r)(p))$
Guarantee $\approx$	$availability(r) := maintenance!$
	$\forall\ b \in\ blocks(r)\ (status(b) := occupied)$
	$route(t) := null$

This specification includes the case in which, although the requested route is available, not all the related blocks have been freed (for example in one block a sensor stopped working). This is a warning situation and the route needs to be put under observation, the train will be assigned to a null route and, for safety reasons, all the blocks in that route will be occupied. An additional layer of R/G has been added for this purpose and it has been indicated by $\approx$.

The "make-it robust" process The process of adding further layers to the specification considering situations that are abnormal (in the sense that they happen less frequently) is called "make-it robust" process and it will be fully developed and formalized as future work. It is out of the scope of this work to explain in detail the formalism behind it, this work represents just an introduction to the method with an explanation of the need for it and its potential application to dependable systems. Anyway, the idea we are working on is to modify the global state, passing from what we call the Ideal World (the initial layer) to what we call the Real World (the further layers, it will never be "real" anyway) according to specific formal rules that have to be applied. In this way we restrict the creative act behind the addition of new layers but we make it possible to automatize the consistency check between different layers. Looking at the Polya's analysis of ancient Greeks problem solving [24], he divides mathematical problems into two classes: "problems to prove" and "problems to find". We have been inspired by this analysis when working on this process. The idea is simply applied: the creative act of identifying the next layer is a "problem to find" and it needs human intervention and invention. This is the hard part of the work. This process is formally guided by a number of rules explaining how the global state, its mappings, the relative domains and ranges and the R/G conditions have to be modified giving a significant spectrum of possibilities, but not infinite freedom. The easy part of the work will be then performed automatically and it will be the "prove" part, the consistency check which represent the automatic correctness analysis.

5.2 The Automotive Example

Now we consider an automotive case study to show the power of the LFTS principle. We will use here a general definition of Cruise control without revealing any confidential information specific of a particular version. For us here a Cruise control (CrCt) is a system that automatically controls the rate of motion of a motor vehicle, the driver sets the speed and the system will take over the throttle of the car to maintain the same speed. In some designs the cruise control has its separate "on/off" switch and controls are easily within the driver's reach (buttons on the wheel generally). Most designs have buttons for "set", "resume", tip up/down". Figure 12 shows a state machine for a simple version of the Cruise Control interface (where there is no tip up and tip down). In the basic version the driver must bring the car up to speed manually and use a button to set the cruise control. At that point, the current speed becomes the desired speed. Most systems do not allow the use below a certain speed discouraging city use. The car will maintain the desired speed but tapping the brake or clutch pedal disables (standby) the system. The idea is that if this does not happen the vehicle would otherwise accelerate against braking to maintain the speed. Some versions can include a memory feature to resume the set speed after braking. The throttle can instead be used to accelerate but once released the car will then slow down reaching the previously set speed.

We use the CrCt to show how the idea of LFTS can be applied in (semi)realistic systems (simplifications of real system for the sake of experimenting with new ideas but still not mere toy examples). Let us consider the following piece of CrCt code :

```
while (target <> current){
   delta := smooth(target, current);
   result := set_eng(delta);
}
```

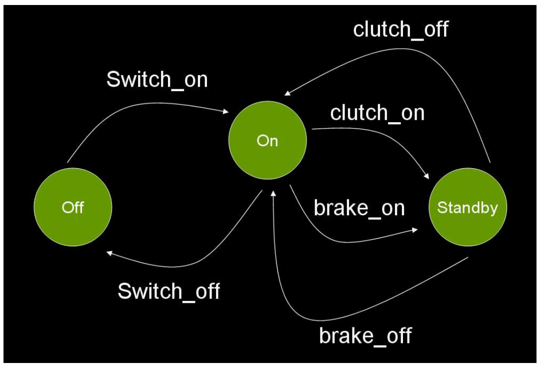

Fig. 12. Simplified State Machine (no tip up/down)

The car speed in acquired in `smooth(target, current)` and then a delta is calculated for the car to have a smooth acceleration (smoothness has to be determined by experience). The specification of this code in term of P,Q,R,G is the following (it is expressed in natural language since we are not giving a mathematical model of the car):

- P: target has to be in a given range
- Q: delta is zero and the driver has been comfortable with the acceleration
- R: the engine is adjusted (smoothly) according to delta
- G: the absolute value of delta is decreasing

Now, one of the usual requirements of a CrCt is to be switched off if an error in engine speed acquisition is detected. This is not taken into account in this specification, when the rely does not hold the engine is not adjusted according to delta. In case the speed acquisition goes wrong the guarantee will not hold and the absolute value of delta will not be decreased. Indeed, following the LFTS principle we should organize it in two layers: a normal mode and an abnormal one (speed acquisition goes wrong):

```
while (target <> current){
   delta := smooth(target, current);
   result := set_eng(delta);
   if result <> OK then
      switch_off
}
```

Now we can add a weaker layer of conditions for the abnormal case being still able to guarantee something. If speed acquisition goes wrong we do not want to force the engine following the delta since it would imply asking for more power when, for example, the car speed is actually decreasing (maybe an accident is happening or it is just out of fuel). Switching the engine off we avoid an expensive engine damage.

6 Conclusive Remarks

In this paper we provided a different view for interpreting problems and faults and we worked toward an improvement of the ideas presented in [20]. Our goal was to start an investigation leading to a method for the formal specification of systems that do not run in isolation but in the real, physical world. To accomplish the goal we passed trough a non trivial number of steps including the discussion of the concept of method itself (computer science has a proliferation of languages but very few methods). Then we presented how we intend to proceed to represent the static and the dynamic view of the problem. A section is dedicated to faults and the following to case studies. We should summarize our method as follows:

1. Define the boundaries of the system/specification (e.g. by the PF approach) - do not specify the universe!
2. Expose and record the assumptions (e.g using RG conditions)
3. Agree 1 and 2 with customer
4. Make it more robust (weakening assumptions)

6.1 Contributions

The main contributions of this work can be considered:

1. An understanding of what a method is and an analysis of the desiderata
2. A formalization of the method in [20] and of the features it has to exhibit
3. Problems described in term of Static View and Dynamic View
4. EI as a model of faults (and consequent introduction of fault tolerant behavior)
5. The organization of the specification in terms of layers of RG conditions (LFTS)
6. The experimentation on practical case studies

6.2 Open Issues

This work is not exhaustive and many aspects need more investigation. Especially the possibility of having Jackson's diagrams extensions working as a bridge between the static and the dynamic view in the way we described them. Thus an open issue is:

Combining static and dynamic views in a coherent and readable notation

Jackson' diagrams extensions are only one of the possible solutions. Another point we just sketched but that needs more work is about the the plug-ins:

Permitting the practical use of different tools/notation

More investigation regarding the case studies is also needed.

Acknowledgments This work has made been possible by the useful conversations with Cliff Jones, Michael Jackson, Ian Hayes, Ani Bhattacharyya, Alexander Romanovsky, John Fitzgerald, Jeremy Bryans, Fernando Dotti, Alexei Iliasov, Ilya Lopatkin and Rainer Gmehlich and it has been funded by the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity) [1]. I also have to thank a number of DEPLOY people that provided me with great feedback during the project meetings.

References

1. Deploy: Industrial deployment of system engineering methods providing high dependability and productivity. `http://www.deploy-project.eu/`.
2. J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.
3. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* To be published in 2009.
4. J.-R. Abrial, S.A. Schuman, and B. Meyer. *A Specification Language.* Cambridge University Press, New York, NY, USA, 1980.

5. J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.
6. K. Bittner. *Use Case Modeling.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
7. D. Bjorner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science.* Springer, 1978.
8. G. Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition).* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.
9. J.W. Coleman and C.B. Jones. Examples of how to determine the specifications of control systems. In A. Romanovsky M. Butler, C. Jones and E. Troubitsyna, editors, *Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems (REFT 2005)*, 2005.
10. P. Collette and C.B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In *Proof, Language, and Interaction*, pages 277–308, 2000.
11. M.L. DeFleur, P. Kearney, and T.G. Plax. Mastering communication in contemporary america. 1993.
12. M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition.* Addison-Wesley Professional, 2003.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 26(1):53–56, 1983.
14. M. Jackson. *Principles of Program Design.* Academic Press, Inc., Orlando, FL, USA, 1975.
15. M. Jackson. *System Development.* Prentice-Hall, 1983.
16. M. Jackson. *Problem frames: analyzing and structuring software development problems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
17. C.B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* PhD thesis, Programming Research Group,University of Oxford, 1981.
18. C.B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
19. C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
20. C.B Jones, I.J Hayes, and M.A. Jackson. Deriving specifications for systems that are connected to the physical world. In *Formal Methods and Hybrid Real-Time Systems*, pages 364–390, 2007.
21. P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.
22. Mannion M. and Keepence B. Smart requirements. *SIGSOFT Softw. Eng. Notes*, 1995.
23. G.D. Plotkin, C. Stirling, and M. Tofte, editors. *Proof, Language, and Interaction, Essays in Honour of Robin Milner.* The MIT Press, 2000.
24. G. Polya. *How to Solve It.* Princeton University Press, 1971.
25. L.J. Lafleur (trans.) R. Descartes. *Discourse on Method and Meditations.* New York: The Liberal Arts Press, 1960.
26. J. Reason. *Human Error.* Cambridge University Press, 1990.
27. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Professional, 1997.