COMPUTER RESEARCH AND MODELING
2024 VOL. 16 NO. 7 P. 1637–1650
DOI: 10.20537/2076-7633-2024-16-7-1637-1650

Ки&М

SPECIAL ISSUE

UDC: 004

Deriving specifications of dependable systems

M. Mazzara

Innopolis University,
1 Universitetskaya st., Innopolis, 420500, Russia

E-mail: m.mazzara@innopolis.ru

Received 28.10.2024, after completion — 12.11.2024
Accepted for publication 25.11.2024

Although human skills are heavily involved in the Requirements Engineering process, in particular, in requirements elicitation, analysis and specification, still methodology and formalism play a determining role in providing clarity and enabling analysis. In this paper, we propose a method for deriving formal specifications, which are applicable to dependable software systems. First, we clarify what the method itself is. Computer science has a proliferation of languages and methods, but the difference between the two is not always clear. This is a conceptual contribution. Furthermore, we propose the idea of Layered Fault Tolerant Specification (LFTS). The principle consists in layering specifications in (at least) two different layers: one for *normal* behaviors and others (if more than one) for *abnormal* behaviors. Abnormal behaviors are described in terms of an Error Injector (EI), which represent a model of the expected erroneous interference coming from the environment. This structure has been inspired by the notion of an idealized Fault Tolerant component, but the combination of LFTS and EI using rely guarantee thinking to describe interference is our second contribution. The overall result is the definition of a method for the specification of systems that do not run in isolation but in the real, physical world. We propose an approach that is pragmatic to its target audience: techniques must scale and be usable by non-experts, if they are to make it into an industrial setting. This article is making tentative steps, but the recent trends in Software Engineering such as Microservices, smart and software-defined buildings, M2M micropayments and Devops are relevant fields continue the investigation concerning dependability and rely guarantee thinking.

Keywords: formal methods, dependability

Citation: *Computer Research and Modeling*, 2024, vol. 16, no. 7, pp. 1637–1650.

Introduction

The material collected in this article has been facilitated by the experience accumulated in working as a software and requirements engineer, working with formal methods and teaching related disciplines at several universities. We have realized that there is a long tradition of approaching Requirements Engineering (RE) by means of formal or semi-formal techniques [Bruel et al., 2022].

In this article, we ask: *how many of the current formal notations used in Requirements and Software Engineering are paired with an effective and clear methodology to apply them*? In our experience, this is the most common obstacle in deploying such ideas in industry (admitting exceptions in specific fields). This article refers directly to the work done in [Jones, Hayes, Jackson, 2007], where the original idea of a formal method for the specification of systems running in the physical world originated.

In Section "Background", we present the background that led to this paper's ideas and approach. In Section "Method", we define a set of desiderata that we believe a method should have and we structure the method introduced by [Jones, Hayes, Jackson, 2007] according to the properties described earlier. In Section "A case study", we present a train case study taken from [Abrial, 2010]. Section "LFTS" introduces the idea of Layered Fault Tolerant Specification (LFTS) and Section "Conclusions" draws final conclusions.

Background

In this section, we present the (philosophical) background that led to our ideas and approaches in this article. More details can be found on an up-to-date survey [Bruel et al., 2022].

Although human skills are heavily involved in the Requirements Engineering process, in particular, in requirements elicitation, analysis and specification, still methodology and formalism play a determining role [Bruel et al., 2022] in providing clarity and enabling analysis. The first thing we realized in building dependable software is the necessity to enable dependable communication between parties that use different languages and vocabulary (i. e., avoid being "lost in translation" between socio-technical domains). In order for systems to match expectations (and specifications), we need a precise mapping between intentions and actions.

Object Oriented Design [Meyer, 2009] and Component Computing [Szyperski, 2002] are well known examples of how rigor and discipline can improve the quality of software artifacts. The success of languages like Java or C# can be interpreted in this sense. 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 and not vice versa.

Semiformal notations like UML [Booch, Rumbaugh, Jacobson, 2005] helped in creating a language that can be understood by both experts and nonspecialists, providing views of the system that can be negotiated between different stakeholders with varied backgrounds. The power and flexibility of UML is also its limitation: the absence of an agreed formal semantics. Attempts of formalizing subsets of UML have never led to a complete and standardized semantics for it [Li, Liu, Jifeng, 2004; Liu et al., 2013].

Pairing UML with OCL [Organization, 2014] makes UML closer to more formal approaches, but it still lacks a fully rigorous notation and method.

Many other formal/mathematical notations existed for specifying and verifying systems like process algebras [Baeten, 2005] or specification languages like Z [Woodcock, Davies, 1996], B [Abrial, 1996] and Event-B [Abrial, 2010]. The Vienna Development Method (VDM) is one of the first attempts to establish a Formal Method [Bicarregui et al., 1994]. All these notations are specific and can be understood mostly by specialists. These formalisms are formal or semiformal notations. Behind each of them there is a way of structuring thinking that does not offer complete freedom and hence forces

designers to adhere to some discipline. Still, they are not methods in the sense described in Section "Method": they are languages.

All these aspects have been discussed in a recently published [Bruel et al., 2022]. The wide spectrum from informal to semiformal and formal are discussed, including methodological considerations. The goal of this paper is instead to focus primarily on the formal parts of that spectrum.

The overall result is 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 go through a number of steps, concepts and tools. The first step, the most important one, is the concept of a method itself, since we realized that computer science has a proliferation of languages, but few methods. In the following sections, we put emphasis on the difference between methods and languages and, as a consequence, between formal methods and formal languages.

Method

This section defines the desiderata for the method. We reached these ideas in an attempt to understand what a method is. Firstly, we think it is important to distinguish between the words method and methodology. The Webster's dictionary says "*a method is a way, technique, or process of or for doing something*". This definition depends on "*a series of actions or operations conducing to an end*". The word methodology can also be used to intend "*a particular procedure*" but the general meaning is "*the analysis of the principles of methods* [. . .] *employed by a discipline*". Although in computer science it is common practice to use the word formal methods to intend formal languages, in this paper we use the word method only to intend the final result of a methodological study related to a specific context: software systems specification.

To properly understand what a method is and what it is not, we explored an example by Descartes [Descartes, 1950]. We are interested in understanding how Descartes perceives a method and what is particular to it. From [Descartes, 1950], we 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 next steps. This suggested to us that the method we propose needs to satisfy a number of properties. The first three are taken directly from Descartes' method, while the last three are from our experiments with case studies. Our understanding of the method of science is that:

1. It has to consist of steps to acquire knowledge;
2. It has to be formally defined (i. e., phases, steps, workflows);
3. It has to be repeatable by nonformal methods experts.

Then our practical experience has suggests that:

1. It has to be scalable (not ad hoc — it has to work outside specific case studies);
2. It needs abstractions (focus on "what" and not "how");
3. It has to be extensible to fault tolerant behaviors.

We have asked ourselves: why do we need a method? The interesting point is a meta question. The logic is what is done inside the system, in this case the formal steps performed (in some order) to reach the desired end. 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. The first step in building dependable software is building dependable communication between parties in different fields that have different languages/vocabulary. According to the definition of communication [DeFleur et al., 2004], formal methods in system

specification are tools to commit to dependability, help clarify vocabulary and provide a notation able to build a precise mapping between intentions and actions in the different stakeholders' minds.

This paper focuses especially on [Jones, Hayes, Jackson, 2007], where the original idea of a formal method for the specification of systems running in the physical world originated. We think an approach that is pragmatic to its target audience is urgent: techniques must scale and be usable by nonexperts if they are to make it into an industrial setting. Thus, the goal of this work is to take the mindset behind the ideas in [Jones, Hayes, Jackson, 2007] accessible and fit them to their target audience industries.

The idea in [Jones, Hayes, Jackson, 2007] is to specify a system not in isolation, but considering its environment and deriving the final specification from a wider system, where assumptions have been understood and formalized as layers of rely conditions. The difference between assumptions and requirements is crucial, especially when considering the proper fault tolerance aspects, as follows:

- Not specifying the digital system in isolation;
- Deriving the specification 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).

This approach allows us to see a computer system from a different angle, as not consisting of functions performing tasks in isolation but as relationships (interfaces/contracts) in a wider context including both the machine and the physical (measurable) reality.

This philosophy has been partly inspired by Jackson's problem frames approach to software requirements [Jackson, 2007]. The machine can only operate through sensors and actuators. We want to derive the specification of the software starting by taking the wider environment of its use into account. We record these assumptions in what follows.

The method and its steps

In this work, we interpret [Jones, Hayes, Jackson, 2007] according to the views described in the previous section by recognizing three main steps:

1. Define boundaries and dependencies of a system;
2. Expose and record assumptions (by means of rely-conditions);
3. Derive a formal specification.

Our idea is to not commit to a single language/notation. These steps are only reference tools that are *suggested* to designers: our main objective is to improve their capabilities through the assistance of a variety of formal notations suitable to each target application. In this work, we want to expand the work done in [Jones, Hayes, Jackson, 2007].

Rely/Guarantee thinking and interference

Rely/Guarantee (R/G) thinking is an idea beyond the formalism. To understand its expressive power, it is necessary to realize how pre- and postconditions can help in specifying a software program when interference does not play a role. What we have to describe are:

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

Pre- and postconditions represent contracts between parties: provided that you (the environment, the user, another system) can ensure the validity of a certain condition, the implementation modify the state in such a way that another known condition holds. We show the example of a very simple program, which in natural language can be:

"*Find the smallest element in a set of natural numbers*"

This 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 ($\mathcal{N}$). The input domain and the output range of the program are then easy to describe:

$$I/O\colon \mathcal{P}(\mathcal{N}) \to \mathcal{N}.$$

Now, the input is expected to be a set of natural numbers, but in order to be able to compute the minimum element, such that a set has to be nonempty, since the minimum function is not defined for empty sets. So, the precondition is that is that the input set is non-empty. This could also be defined through the function type as:

$$I/O\colon \mathcal{P}_1(\mathcal{N}) \to \mathcal{N}.$$

Provided that the input is a nonempty set of natural numbers, the implementation is able to compute the minimum element, which is the one satisfying:

$$Q(S, r)\colon r \in S \wedge (\forall e \in S)(r \leqslant e).$$

That is, for any input set S the result r must be the smallest element in that set. Given this rule, the input-output relation is given by the following predicate that needs to be satisfied by any implementation f:

$$\forall S \in \mathcal{P}_1(\mathcal{N})(P(S) \Rightarrow f(S) \in \mathcal{N} \wedge Q(S, f(S))).$$

This is an example of a proof obligation associated with the design to be discharged later by the specifier.

To better understand the limitations of this kind of abstractions, consider the case of an implementation with interference happening through a global state, where two processes alternate their execution and access to the global state. It can consist of shared variables or can be a queue of messages, as described in [Jones, 1983]. In case 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 constraints to these changes. 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 R (rely) and the process under analysis can change its state only in such a way that observations by other processes 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. For example, the two following concurrent pieces of code calculates the Greatest Common Divisor:

```
P1:                 P2:
while(a<>b){        while(a<>b){
  if(a > b)           if(b > a)
    a := a-b;           b := b-a;
}                   }
```

`P1` is in charge of decrementing a, and `P2`, of decrementing b. When $a = b$ evaluates to true, it means that one is the Greatest Common Divisor (GCD) for a and b. The specification of the concurrent interfering interactions is as follows:

$$R_1 : (a = \overline{a}) \wedge (a \geqslant b \Rightarrow b = \overline{b}) \wedge (GCD(a, b) = GCD(\overline{a}, \overline{b})), \quad R_2 = G_1,$$
$$G_1 : (b = \overline{b}) \wedge (a \leqslant b \Rightarrow a = \overline{a}) \wedge (GCD(a, b) = GCD(\overline{a}, \overline{b})), \quad G_2 = R_1.$$

Here the values $\overline{a}$ and $\overline{b}$ are used instead of a and b when we want to distinguish between the values before and after the execution. `P1` relies on the fact that `P2` is not changing the value of a and that ($a \geqslant b$) means no decrements for b have been performed. Furthermore, the GCD did not change. What is a guarantee for `P1` becomes a rely for `P2` and vice versa.

A case study

In this section, we show a case study in which we applied the method discussed earlier. It is taken from [Abrial, 2010], *the train system*, where the goal was to show the power of modeling and formal reasoning through Event-B. We chose this scenario since we believe it is 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 [Jones, Hayes, Jackson, 2007]. We show here how this example can be approached with the three step method. The first thing to do is determining the bounds of specification (Step 1). We then show how the boundaries can be broadened to include the external world. In the second step, we discuss how to separate assumptions and requirements, how to expose and record assumptions and how different sets of requirements and assumptions imply a different specification and implementation. In the third step, we assume the existence of an already designed network infrastructure (with sensors, actuators, *etc.*) to show a specific example of an implementation. At the end, we show how to make use of rely conditions for this specific implementation. What is important to realize is the way in which the interference over a global state is considered using the approach shown in the small GCD example. The specification is presented after a discussion about the way in which it has been obtained, and the interaction between the different operations constrained in a similar way, but in a system with a potentially higher level of concurrency.

Step 1: defining the system's boundaries

First, clarify the real-world requirements before trying to specify their software system. This process naturally identifies assumptions about the physical components, which can then be recorded as rely-conditions. One of the main principles of this approach is not specifying a system in isolation, but starting to move the system boundaries outwards; what is called "pushing out the boundaries of the system" in [Jones, Hayes, Jackson, 2007]. What is the wider system in which physical phenomena

are measurable? What is the actual general purpose of the Train System? We believe it allows trains to move safely from a place X to a place Y. How does this help us in identifying the system requirements? We can recognize that the FUN-1 requirement of the system specification [Abrial, 2010] expresses this need as:

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

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

- Safety property: nothing bad can happen;
- Liveness property: something good has to happen.

We can express these two properties for this example as:

- 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. We are interested in modeling only the safety property, hence delegating liveness to a scheduler or, theoretically, to manual management performed by operators/engineers. All other requirements in [Abrial, 2010] 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.

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 of the requirements can be considered as assumptions taken from the already existing environment (for example, the ENV group of requirements in [Abrial, 2010]). Otherwise, they can still be seen as requirements, but referring to a specific implementation. 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 how requirements and assumptions can be (in our opinion erroneously) mixed 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. The simplest possible implementation of this requirement is that we do not allow any train to cross the network at any time. This is an implementation where the Safety property is preserved (but the Liveness property might not). Although we are interested mainly in the Safety property, a better thing to do is to allow only one train on the track between X and Y. This means that the rail connecting two cities is reserved for a single train. Obviously, this implementation respects both the Safety and Liveness properties described above. Nevertheless, it is easy to realize that it is simply unfeasible because of the low efficiency/exploitation of available resources.

An alternative implementation is the one in [Abrial, 2010]. 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 its entire discussion. The point was learning the lesson about determining wider boundaries, including the external environment, and distinguishing between requirements and assumptions. Figure 1 represents an example of the infrastructure. It is made of:

1. Blocks: a track is made of a number of fixed blocks;
2. Routes: blocks are always structured in a number of statically predefined 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 consisting of blocks LABDKJN is possible in Fig. 1.
3. Points: a track contains special components allowing blocks to be linked to each other. 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 Fig. 1, B and D both contain points, and C does not.
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.

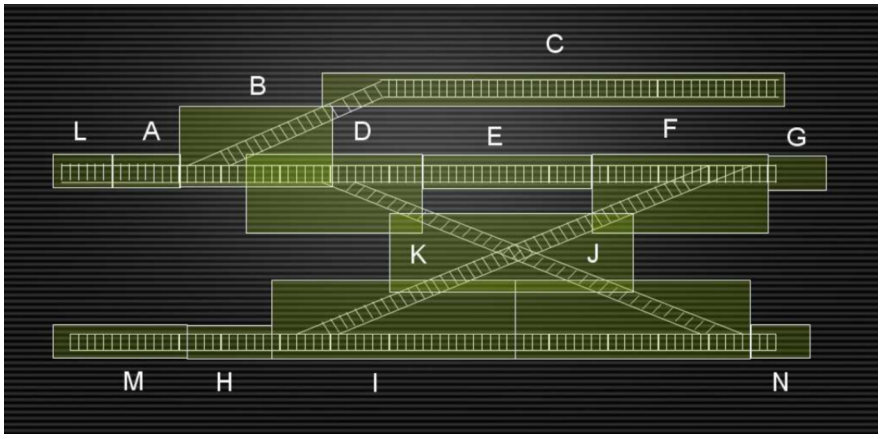

Figure 1. The network infrastructure. Green boxes represent Blocks of which a track is made and they are indicated with capital letters. Blocks B, D, F, K, I and J contain points

The idea is to have each block of a reserved route freed as soon as the train does not occupy it anymore. In the next section, we focus on the reserving routes system, i. e., the process of reserving 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, since the underlying block-based infrastructure ensures that we never have two trains in the same block to avoid collisions. 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 consists of the sensors and actuators necessary to make such an infrastructure work. We have also decided to focus on Safety properties. We assume that Liveness properties are managed by a scheduler, which is another system already running. A graph structure would be probably more adequate it we wanted to focus on the scheduler having Liveness coming into play, since in the present representation the network is seen as a set of routes from which you cannot infer which one is adjacent to the other.

Step 3: deriving the formal specification

Now we define the basic machinery for the formal specification. We need four (non-empty) finite sets as:

- T, a finite set of trains, where $t \in T$;
- B, a finite set of blocks, where $b \in B$;
- R, a finite set of routes, where $r \in R$;
- P, a finite set of points, where $p \in P$.

The safety requirement is modeled as a total function mapping blocks to trains: $\mathtt{B} \to \mathtt{T}$ (`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: $\mathtt{T} \to \mathtt{R}$ (`route`). A route is then composed by blocks, at least one: $\mathtt{R} \to \mathtt{B}^+$ (`blocks`) and in a route a block has the next element: $\mathtt{B} \to \mathtt{B}$ (`next`). Blocks can be free or occupied: $\mathtt{B} \to \{free, occupied\}$ (`status`) and are associated to points: $\mathtt{B} \to \mathtt{P}$ (`point`) that can be oriented in two different ways: $\mathtt{P} \to \{directed, diverted\}$ (`direction`). Routes can be available or reserved: $\mathtt{R} \to \{available, reserved\}$ (`availability`) and each route is associated with a predefined points orientation: $\mathtt{R} \to (\mathtt{P} \to \{directed, diverted\})$ (`orientation`). We rely on the fact that the sensors with which a block is equipped can always detect the presence of a train (for $\mathtt{B} \to \mathtt{T}$). We assume that if we want to reserve a point, it is promptly positioned. We rely also on the fact that each route has the first block: $\mathtt{R} \to \mathtt{B}(first)$, the last block: $\mathtt{R} \to \mathtt{B}(last)$, and that 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 define operate: they are related to the process of route reservation and freeing plus the entrance, proceeding and exiting 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 were previously organized. For each operation, the notation below indicates the data needed, what we expect from that data and the way in which the global state is modified.

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

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

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

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	$availability(route(t)) = reserved \wedge$
	$\wedge\ b \in blocks(route(t)) \wedge$
	$\wedge\ next(b) = \varnothing$
Guarantee	$\forall b \in blocks(route(t))\ status(b) := free$

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

The previous sections discussed how to derive a specification of a system looking at the physical world in which it is going to run in. No mention has been made of fault tolerance and abnormal situations that deviate from the basic specification. The method discussed in Section “Method” define the three steps one has to follow to specify a system and they do not depend on what you are actually specifying, or on its fault tolerance requirements. 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 more robust system. The same can be done by weakening rely conditions. For example, if eliminating a precondition the system can still satisfy the requirements, this means we have a more robust system. Here we promote this approach considering the idea of fault as interference. Quoting [Collette, Jones, 2000]:

> “The essence ... 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 considered.”

In this work, we introduce the idea of Layered Fault Tolerant Specification (LFTS) combining it with the approach quoted above and making use of rely/guarantee thinking. The first step in this direction is defining a Fault Model, i. e., which kind of abnormal scenarios we are considering. Our specification then takes into account that the software runs in an environment when specific things can behave in an unexpected way. There are three main abnormal situations they can incur:

- Deleting state update: “lost messages”;
- Duplicating state update: “duplicated messages”;
- Additional (malicious) state update: “fake messages created”.

The first one means that a message (or the update of a shared variable) has been lost, i. e., its effect is not taken into account, as if it had 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 because of hardware, middleware or software malfunctioning). In this case a fake message (or update) is created from scratch containing unwanted information.

In our approach, the model of a fault is represented by an *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 concepts discussed in [Moradi et al., 2018]. Here, we only mean a model of the erroneous behavior of the environment. This behavior is limited, depending on the number of abnormal cases we intend to consider, and the EI always plays its role by respecting the defined R/G rules. The operations rely on a specific abnormal behavior and guarantee the ability to handle these situations. The extended R/G rules are as follows:

- The Error Injector (environment) interferes by changing the global state, but respecting his G. For example, only lost messages can be handled;
- The operation relying on this kind of (restricted) interference is able to handle exceptional/abnormal (low frequency) situations satisfying a weaker G.

All the possibilities of faults in the system are described in these terms and the specification is organized according to the LFTS principle of layering the specification, for the sake of clarity, in (at least) two different levels, the first one is for *normal* behaviors and others (if more than one) are for *abnormal* behaviours. This approach originated from the notion of idealized fault tolerant component [Anderson, Anderson, Lee, 1981], but the combination of LFTS and R/G reasoning is the main contribution of this work. From the expressiveness point of view, a monolithic specification can include all the aspects, both faulty and nonfaulty, 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, by following the LFTS principles a specification, can become understandable for all the stakeholders involved.

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 the third

option: R → {*available*, *reserved*, *maintenance*!} (`availability`). The *RouteReserving* operation can be extended as follows:

Operation	*RouteReserving* (t : T, r : R)
Rely	$availability(r) = available \wedge$
	$\wedge\ \forall b \in blocks(r)\ (status(b) = free)$
Rely ≈	$availability(r) = available$
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 ≈	$availability(r) := maintenance!$
	$\forall b \in blocks(r)\ (status(b) := occupied)$
	$route(t) := null$

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 is assigned to a null route and, for safety reasons, all the blocks in that route are occupied. An additional layer of R/G is added for this purpose and it is indicated by the ≈ syntax.

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 requires further analysis and formalization. It seems to be beyond the scope of this paper to explain in detail the formalism behind it since the article 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 will always be an abstraction of Physics), 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 [Polya, 1971], 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 to give a significant spectrum of possibilities.

Conclusions

In this article, we have collected reflections and worked toward an improvement of the ideas presented in [Jones, Hayes, Jackson, 2007]. The main contributions of this paper are:

1. A better understanding of what a method and desiderata analysis are;
2. Formalisation of features described in [Jones, Hayes, Jackson, 2007];

3. EI as a model of faults for capturing fault tolerant behaviors;
4. The organization of specification in terms of layers of R/G conditions (LFTS);
5. The experimentation on a practical case study.

Although the article only makes tentative steps, the hope is that this may revive the discussion initiated in [Jones, Hayes, Jackson, 2007] and bring a future flow of new literature on this topic. The recent trends in Software Engineering such as Microservices [Bucchiarone et al., 2020], smart and software-defined buildings [Mazzara et al., 2019], M2M micropayments [Strugar et al., 2018] and Devops [Bobrov et al., 2019] are also a relevant field of investigation of what concerns dependability and rely guarantee thinking. To the best of our knowledge, not much literature has been published in this area.

References

Abrial J.-R. Modeling in Event-B: system and software engineering. — Cambridge University Press, 2010.

Abrial J.-R. The B-book: assigning programs to meanings. — Cambridge University Press, 1996.

Anderson T., Anderson T., Lee P.A. Fault tolerance, principles and practice. — Prentice/Hall International, 1981. — https://books.google.ru/books?id=LtdQAAAAMAAJ

Baeten J. C. M. A brief history of process algebra // Theoretical Computer Science. — 2005. — Vol. 335, No. 2–3. — P. 131–146. — http://dx.doi.org/10.1016/j.tcs.2004.07.036

Bicarregui J. C., Fitzgerald J. S., Lindsay P. A., Moore R., Ritchie B. Proof in VDM: a practitioner's guide. — Berlin, Heidelberg: Springer-Verlag, 1994.

Bobrov E., Bucchiarone A., Capozucca A., Guelfi N., Mazzara M., Masyagin S. Teaching DevOps in academia and industry: reflections and vision // Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment — Second International Workshop, DEVOPS 2019, Château de Villebrumier, France, May 6–8, 2019, revised selected papers / Eds.: J.-M. Bruel, M. Mazzara, B. Meyer. — Springer, 2019. — Lecture Notes in Computer Science. — Vol. 12055. — P. 1–14.

Booch G., Rumbaugh J., Jacobson I. Unified modeling language user guide. — Addison-Wesley Professional, 2005.

Bruel J.-M., Ebersold S., Galinier F., Mazzara M., Naumchev A., Meyer B. The role of formalism in system requirements // ACM Comput. Surv. — 2022. — Vol. 54, No. 5. — P. 93:1–93:36. — https://doi.org/10.1145/3448975

Bucchiarone A., Dragoni N., Dustdar S., Lago P., Mazzara M., Rivera V., Sadovykh A. (eds.) Microservices, science and engineering. — Springer, 2020.

Collette P., Jones C. B. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations // Proof, language, and interaction, Essays in Honour of Robin Milner / G. D. Plotkin, C. Stirling, M. Tofte. — The MIT Press, 2000. — P. 277–308.

DeFleur M., Kearney P., Plax T., DeFleur M. Fundamentals of human communication. — McGraw-Hill Companies,Incorporated, 2004. — https://books.google.ru/books?id=2EoPAAAACAAJ

Descartes R. Discourse on method. — Harmondsworth, Penguin, 1950.

Jackson M. The problem frames approach to software engineering // 14th Asia-Pacific Software Engineering Conference (APSEC'07). — 2007. — P. 14. — DOI: 10.1109/ASPEC.2007.11

Jones C. B. Tentative steps toward a development method for interfering programs // ACM Trans. Program. Lang. Syst. — 1983. — Vol. 5, No. 4. — P. 596–619. — https://doi.org/10.1145/69575.69577

Jones C. B., Hayes I. J., Jackson M. A. Deriving specifications for systems that are connected to the physical world // Formal Methods and Hybrid Real-Time Systems, Essays in Honor of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays, Papers presented at a Symposium held in Macao, China, September 24–25, 2007 / Eds.: C. B. Jones, Z. Liu, J. Woodcock. — Springer, 2007. — Lecture Notes in Computer Science. — Vol. 4700. — P. 364–390. — https://doi.org/10.1007/978-3-540-75221-9_16

Li X., Liu Z., Jifeng H. A formal semantics of UML sequence diagram // Proceedings of the 2004 Australian Software Engineering Conference. — 2004. — P. 168.

Liu S., Liu Y., André É., Choppy C., Sun J., Wadhwa B., Dong J. S. A formal semantics for complete UML state machines with communications // Proceedings of Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10–14, 2013. — Springer, 2013. — Lecture Notes in Computer Science. — Vol. 7940. — P. 331–346.

Mazzara M., Afanasyev I., Sarangi S. R., Distefano S., Kumar V., Ahmad M. A reference architecture for smart and software-defined buildings // IEEE International Conference on Smart Computing, SMARTCOMP 2019, Washington, DC, USA, June 12–15, 2019. — 2019. — P. 167–172.

Meyer B. Touch of class: learning to program well with objects and contracts. — Springer Publishing Company, Incorporated, 2009.

Moradi M., Van Acker B., Vanherpen K., Denil J. Model-implemented hybrid fault injection for Simulink (tool demonstrations) // Cyber physical systems. Model-Based Design — 8th International Workshop, CyPhy 2018, and 14th International Workshop, WESE 2018, Turin, Italy, October 4–5, 2018, revised selected papers / Eds.: R. D. Chamberlain, W. Taha, M. Törngren. — Springer, 2018. — Lecture Notes in Computer Science. — Vol. 11615. — P. 71–90.

Object Management Group Standards Development Organization. Object constraint language. — [Electronic resource]. — 2014. — https://www.omg.org/spec/OCL/ (accessed: 14.02.2024).

Polya G. How to solve it. — Princeton University Press, 1971. — http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0691023565

Strugar D., Hussain R., Mazzara M., Rivera V., Lee J., Mustafin R. On M2M micropayments: a case study of electric autonomous vehicles // IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 – August 3, 2018. — 2018. — P. 1697–1700.

Szyperski C. Component software: beyond object-oriented programming. — Addison-Wesley Longman Publishing Co., Inc., 2002.

Woodcock J. C. P., Davies J. Using Z: specification, refinement, and proof. — Prentice Hall, 1996.