Dependability Assessment of SOA-based Cyber-Physical Systems with Contracts and Model-Based Fault Injection

Autore: Loris Dal Lago

Curatori: Orlando Ferrante, Roberto Passerone

February 2017

Technical Report # DISI-17-003

UNIVERSITÀ DEGLI STUDI DI TRENTO
DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE

Master Degree in Computer Science

Final Thesis

Dependability Assessment of SOA-based Cyber-Physical Systems with Contracts and Model-Based Fault Injection

Advisor: Roberto Passerone

Co-advisor: Orlando Ferrante

Candidate: Loris Dal Lago

ACADEMIC YEAR 2014/2015

To my family.

Acknowledgements

This thesis is the outcome of an internship in ALES (Advanced Laboratory on Embedded Systems). Mainly I am grateful to Orlando Ferrante for giving me the opportunity to work on this project and for being supportive all throughout the period of my stay at the company. I also want to thank professor Roberto Passerone, firstly for being my advisor and second for making himself available for help whenever I was asking him for. My understanding of contracts in the first place would not have been as clear without his help. Also I would like to express my gratitude to Thi Thieu Hoa Le for her positive karma in giving help to people: her technical assistance helped me in more than one occasion to sort things out. Thank you Hoa. These were the people to whom I partly owe the scientific work of this thesis.
I cannot conclude this fragment though without acknowledging the immense support, mostly moral, that I received from my friends and family throughout my years at the university, in my country and when I was abroad. I want to thank my mum, dad, grandmother and sis, because you have always been the best people around, cheering and inspiring without even knowing. Thanks to all my closest friends too, for giving rest to my unconditional study efforts. That has been so vital. Lastly, I would like to send my thanks to all the people who have been part of my study career, including also those who can smile to the words: *Loris doesn't like beauty.*

Abstract

In the era of interconnected systems it is becoming more and more important to address issues of scale in system safety and dependability assessment. Concurrency has no longer to be understood only as a concern *within* individual systems - but rather also and mostly as *between* them.

Yielding the correct deployment of such systems is thus challenging. Besides technical complications also social collaborations between engineering teams intertwine, whereby the need of communicating unambiguously their component's potentials and needs is both paramount and difficult to attain. This is a major concern in the development of Cyber-Physical Systems (CPSs), where computation is put at service of the physical world to let constituent embedded components accomplish some given goal.

The main objective of this thesis is the development of novel techniques for the dependability analysis of highly distributed systems structured over the concept of Service-Oriented Architecture (SOA), with particular consideration to needs coming from the industry sector. Focusing on component services as the central means at the core of each interaction, our methodology will thus show how existing techniques applicable to plain digital components can be lifted to the broader area of complex Cyber-Physical Systems.

We will commit on modeling Cyber-Physical Systems of different sorts, first accounting for their cyber-physical aspects and then considering changes in topology typical of SOA. For this we will stick to the UML language extensions of SoaML and SysML, being them particularly amenable to industry. We will then advocate the intensive use of contracts, already well-established for the design of heterogeneous CPSs, as an indispensable and perfectly tailored concept for services. We will demonstrate how non-functional requirements, such as timing, can be incorporated in the presented framework.

At the end of the presentation we will demonstrate how the employment of XSAP, a tool for the safety analysis of state-based systems based on nuXmv, can be adapted to accommodate the dependability analysis of SOA-based Cyber-Physical Systems. To that aim we will show how to model and analyze two specific use case examples inspired by the literature by featuring XSAP in a novel way. We will present the results of the analysis with respect to our findings and conclude with a discussion over the whole methodology and future directions.

Contents

Introduction

The design of behaving computing systems without the employment of automatic verification tools is nowadays simply not a viable option. The complex interactions and high concurrency typical of those systems make the employment of these tools, in fact, obligatory.

Model checkers, intended as systems based on the verification of state machines, have been in the past and still are the majorly employed solution in industry[32]. For the embedded systems community in particular, symbolic model checkers have been pivotal for the verification of systems with an enormous number of states, that were before insurmountable. Finally, the introduction of bounded model checking techniques gave the final strike for the adoption of model checkers in industry to even infinite systems, providing a technique that, albeit incomplete, could quickly find bugs, alias property counterexamples, in the system models.

Despite the nice properties that come with model checking, there is a big gap that makes them not entirely congruent with the embedded systems' demands, given by the abstraction-level/implementation separation. For how close a model of the system might be to reality, it will never be able to capture completely all of its little details. And there is more. Today's computing systems' are made up of different heterogeneous components, combined to reach goals collaboratively. An automobile comprising an ABS controller and a power steering system represents one such collaborative interaction between two components. Most failures in such collaborating systems, it turns out, do not happen because of single components' failures, but rather because of unintended interactive actions between components. This fact is understood by acknowledging that, while single components are responsibility of single distributors — thus mostly much robust and prone to reuse — the design of a system that exposes those components as interactive units is constructed ad-hoc and depends, heavily, on how well the system description and requirements are captured first and interpreted by the design and system engineers next. This segment of the system development cycle, where the human interaction has its larger influence, is the weakest part of the process and studied as an engineering branch by itself [68]. This weakness is the basic reason why system verification is always accompanied by — fused with, more recently — system validation[1].

Cyber-Physical Systems (CPSs) represent a class of complex computing systems that are typically deployed on the physical world and tightly integrated with it, in a collaborative way. Characteristically, CPSs are built of heterogeneous components in a flexible structure topology. This means that components are independently developed, using different techniques and tools, by different people, with bindings that are not known in advance, before deployment.

Integration of such component systems is a challenge by its own, with the specific scheme typically — but not quite lately — agreed beforehand, when the overall system is conceived. If all the specifics are known in advance and the system architects are in position to create a standard for the specific application, the simplest thing to component integration is using predefined interfaces, purely as methods' signatures that the single components are explicitly required to match. This has been at the essence of the Remote Procedure Call (RPC) paradigm since the early distribution of computing power on networks. However, along with the recognition of Cyber-Physical Systems as a new-standing concept, a more principled and structured approach of communication is needed.

Evolving the RPC paradigm, Service Oriented Architectures (SOAs) put services at the heart of systems, rather than components with their functionality. This shift has been inspired by web architecture design practice, in some application scenario even integrated with it.

We will present the modeling of Cyber-Physical Systems as SOAs, where the single components constitute SOA participants, requiring services from one another. Service orientation is becoming the new trend in the field, given its generality, several-year groundwork and disposition towards handling of dynamicity. For their distributed nature it will be easier for us to treat the communications between components as service choreographies, without a central control. This will also be convenient for having a global system view and promote its verification.

The service-oriented nature of the systems that we are going to consider

[1]To understand the difference between verification and valudation, it is commonly proposed the following pun as an answer: Verification answers the question: Are we building the product right?, whereas Validation answers to: Are we building the right product?

will bring us to the restatement of these architectures in terms of contracts, a popular formalism in the cyber-physical community for describing, neatly, functional behaviours of the system participants in terms of their assumptions from the environment and provided guarantees. We will talk about methods for the dependability assessment of systems and circuits by means of fault injection and eventually show how those techniques have gradually taken a more and more abstract appeal for the principled adoption since the earliest stages of system development.

Before discussing the themes of this thesis in detail, we outline its driving motivations, planned objectives and realized contributions, in order to delineate the frame where this thesis finds context. We then introduce the structure of the presentation, as an orientation for the reader.

Motivation

Developing reliable systems has always been a matter of concern for industry. This problem is commonly tackled by the adoption of formal verification methods, which are although known to be insufficient to release trustworthy dependable systems. In order to address dependability it is necessary to understand how systems can deviate from the normal behaviour, actuating unexpected actions, and how the system should respond to those. Traditionally, hand-crafted artifacts that outline this behaviours are developed as the result of a combined effort between safety and design engineers. These artifacts comprise Fault Trees and FMEA tables, that are visualizations of how a system can fail in the wake of fault occurrences [24]. In the past decade, new automatic constructions of theses artifacts have been proposed, mainly supported by model-checking activities.

Our work finds its motivations in studying SOA-based Cyber-Physical Systems in context of their dependability assessment using techniques similar to those employed in the past for the digital system context. Challenges will be the large-scale, the non-functional aspects of those systems and the possibility of acceptance by the industry sector.

Objectives

Programmed for 2015, Fondazione Bruno Kessler (FBK) releases a tool for the support and integration of the engineering activities of system design and safety assessment. This tool, named XSAP, builds on top of the NuSMV/nuXmv model checker and features a wide range of functionalities from the system verification to the automatic generation of fault trees, passing through a model-based variant of fault injection that we will recurrently call *model extension*. We will discuss fault injection practice and the tool XSAP in detail in chapter 1 and 4, respectively. The intended support provided by the XSAP tool has been driven by needs from the strict Embedded Systems engineering area, for the safety analysis of digital circuits and Cyber-Physical digital circuits.

Our objectives cover a broader application domain that is pushed by European-sponsored industrial activities, that have the analysis of large distributed systems as target, comprising their architectures and design principles, with particular regard to scalability issues. We focus on Service-Oriented distributed Cyber-Physical Systems. Keeping scalability and service-orientation at the center of our concerns, our objective will thus be to understand the context and its intrinsic dynamics, find out ways for the modeling in industry and delineate systematic methodologies to address automatic dependability resources, culminating in an activity workflow for the engineering of large-scale systems to take on, based on tool automation for how much as it is meaningful to conceive.

Contributions

The contributions of this thesis go from a theoretical comprehension of the context of Cyber-Physical Systems to the proposal of novel practical activities and modeling techniques for their dependability analysis, motivated by the arguments that we summarized above. Moved by the objectives outlined in the last section we probe the validity of our techniques by practical implementations of the suggested techniques on simple use case examples.

Our contributions include showing what languages are available to model such systems at the industry side of concerns, which ones are convenient and which one are not yet mature beyond the academic walls, ending up in a proposed adoption the UML language derivatives of SysML and SoaML for system design, compliant with our approaches and modeling needs, still applicable in the industry practice.

We showed how contracts can involve in service orientation and how much conforming is their adoption with respect to existing literature formalisms. With their power at hand we showed how they can support the modeling of non-functional properties, such as timing, in a consistent way with the rest of the methodology. After modeling two simple use case examples in SysML+SoaML first and nuXmv therefore, the safety analysis tool XSAP is used for the assessment of their dependability using model-based fault injection. Our contribution in this sense is showing its application on systems other than digital. Moreover we will propose a novel and simple approach that exploits duality in XSAP in order to infer activities for system design and planning for free. Open questions throughout the exposition aiming at future work constitute an important part of our work.

Organization of the thesis

The thesis is divided in two parts. The first part covers the state of the art and effective background needed throughout the second part. This will be concerned with the construction of novel design and dependability assessment techniques for the development of highly complex SOA-based Cyber-Physical Systems.

Before analyzing the techniques proposed by this thesis, we will spend a few words discussing where the branch of fault injection stemmed from originally, and how we are going to use this term against its various meanings. This will be subject of chapter 1, where we survey fault injection techniques and stress on their effective need in industrial concerns.
After that, in chapter 2 we will detail about SOA and discuss existing modeling and verification trends on those systems, with an eye kept on the Cyber-Physical basis that they will have to be compliant with according to our objectives. From this analysis of the literature we will identify a number of interesting features that we aim at handling, including scalability and interface behaviour of components in the Cyber-Physical scenario.
Following those lines, in chapter 3 we will make considerations about the adoption of a lately developed mathematical framework and design paradigm for the the development of complex systems. We will also discuss its integration as a basis for our analysis. It will be our objective there to provide the reader with a feeling of legitimacy in the adoption of the framework.
Finally, in chapter 4 we will present the tool XSAP, devised for (although not restricted to) the automatic construction of engineering artifacts in the context of digital systems and utilized in the second part of the thesis in the context of SOA-based Cyber-Physical systems. Here we will describe the algorithmic availabilities of the tool and the artifact's construction procedure. At the end we will outline the pragmatic prerequisites for using the tool, its input language in particular.

From there it will stem the second part, which builds on the in-depth description of the techniques proposed in this thesis. We will indicate two derivations of UML, namely SysML+SoaML, as a modeling tool amenable to industry and then present two use case examples taken from the literature in chapter 5. Those same use cases will be used in chapter 6, when we will finally present the core of our contributions by describing our scheme, developing the analysis using applications of XSAP in a novel way. We will summarize the gist of the thesis and outline our programmed future work, eventually, in the last chapter of the thesis.

Part I

State of the Art

Chapter 1

Fault Injection

This chapter offers a chronological journey in the realm of Fault Injection. We present it as a few survey glances in the literature for approaching the subject and appreciating how this methodology could evolve with time through application domains.
As we will see, the original understanding of the subject founded on the dependability assessment of hardware systems and electrical circuits, only later moving to the injection of programming errors in software and up to model fault injection, inspired by that line. The first and second trends are presented, respectively, by the two survey papers that we revisit as a start. After that we consider a more recent tendency to assess complex systems' dependability that uses logical abstractions of system's details and model-checking techniques in combination with the injection of faults. This will be the case that we will focus on in the second part of the thesis, therefore we will spend some more words for them.
A clear distinction between the three cases hinted above is then outlined and an example of a wrong conception is put forward. The recent development of Contract Based Safety Analysis for component-based Embedded Systems is detailed and new prospects indicated. For all the above instances relevant scientific works are discussed and alternative side cases illustrated.

1.1 A first quick overview of FI techniques

Fault injection in its original flavour developed as the electrical engineering practice to corrupt signals at the hardware level in order to simulate the occurrence of faults in the system and test its robustness against. This practice, widely known as *Hardware Fault Injection*, has then been forwarded to the software layer to corrupt registers' or memory's bit values via hardware instructions, obtaining a less effective but more broadly applicable method that eludes the risk of compromising the hardware physically. This later method is known in the literature as *Software Implemented Hardware Fault Tolerance* (SIHFT), or sometimes with the simpler and less evocative name of *Software Fault Injection* (see [88] for a tutorial). Further developments in this direction use *Simulation-Based Fault Injection* (primarily through VHDL models) and *Emulation-Based Fault Injection* (using FPGAs) to assess system robustness, with the aim of being faithful to the hardware architecture and timing model.

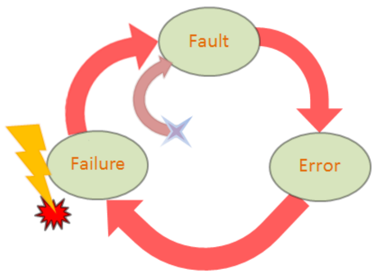
The topic is thoroughly surveyed in [96] where a rigorous presentation is followed, highlighting pros and cons of a wide range of tools. In that work the authors also identify the necessary requirements for a fault injection environment to be such. In particular they find that all approaches in this area, no matter the modality, feature:

1. a *fault injector* to perform the task
2. a *library of admissible faults* for the system to pull out and inject
3. a *workload generator* of admissible system inputs and
4. a *a monitor*, that collects runtime information and displays summary statistics at the end.

Noticeably those four components have been a constant in all types and variants of Fault Injection since, always showing up in one way or another as distinctive features.

Lately, the technique of Fault Injection has been extended to more abstract frameworks in order to test, in particular, software and hardware logically, by injecting faults in their abstracted semantics. This trend is unexplicitly presented in [60], where system dependability is examined as a compound of *reliability* (correct, continuous performance of the system, without failures), *availability* (continuous operation regardless of system failures), *safety* (correct detection of system faults and ability to have appropriate countermeasures), *integrity* (non-corruption of data in time) and *maintainability* (ease to fix the system in response to a failure). Security is not considered here as part of the dependability picture and seldom has been in the field.

A distinction is made in [60] between fault *prevention*, *tolerance*, *forecasting* and *removal*. Stress is put in particular on tolerance - i.e. the ability to tolerate faults during system execution - being it what most systems base their dependability assessment on. In that work, and in most of the literature as well, *faults* are regarded as physical defects or imperfections that lead, possibly, to *failures* (in the sense of expected behaviour). The transit from faults to failures is normally conceived as through deviations from the expected behaviour called *errors*. Failures, if not contained, may in turn trigger other system faults and lead the system, cyclically, to an unmanageable disaster.



However, despite the great popularity that the cyclic causality model has been acknowledged of in the past, today's Cyber-Physical Systems appear not to be following it anymore. What instead happens is that, given the strongly heterogeneous, intertwined and multi-functional characteristics of these systems, wrong specifications/interpretations of both functional and non-functional system requirements by the development parties, lead to nasty failures that are not due to behavioral issues of components, rather else to intricate interactions between them [46][94]. The cyclicity of the loop above is thus broken and a whole lot of unpredictability gets into the picture, calling for a different modeling of faults and failures, that may be triggered independently without the need of a causality event chain.

As a response to this necessity another approach has been proposed, that starts palely from [22] and [23], where the tool FSAP/NUSMV-SA is presented. The idea at its basis is to rise of one level of abstraction and inject faults in the model specification instead than inside the actual system implementation. As an enrichment of the standard specification procedure, requirements are listed according to how the system is expected to react to faults and then checked against the formal model. Requirements in this sense may ask, using temporal modalities, whether after a state is hit by a fault, the system always gets to safe states that respond to the fault in some appropriate way. Moreover, the system also supports the automatic generation of *Fault Trees* (which are causality trees) and offers *Fault Traceability* capabilities to track how and in what order the system may be led to failure. This is done through extensions of standard counterexample-based model-checking techniques together with minimality requirements, using reachability procedures and BDD-based minimization of boolean functions[42][21].

Noticeably this approach to fault injection is design-oriented. It is not its philosophy to assess dependability of the finalized system as the previous approaches used to, rather it finds a broader application at the early stages of system development and right through to the end system, virtually providing support to all system-design phases.
As a matter of truth, the system in [23] was not yet mature for confronting itself with the big challenges of the aforementioned today's design activities, but it has certainly to be given credit for taking the first steps in that direction. The model is still quite faithfully cyclic and causal as in the picture above, it uses fault trees to carry out the analysis. However, although an overall view of the system as a whole is not present, only the behavioral model is considered for the injection. Unlike the earlier approaches to fault injection, [23] runs the causality model on the *logic* of the component, not on its software implementation, not on its hardware. The result of this is that it combines model checking to Fault Injection, in a novel way.

The work in [23] gave rise to a number of efforts in the research community to provide model-based means for dependability analysis, majorly from the same research group, but not only. It is worth mentioning in fact the research activity run by Ezekiel et al. [49][50] on epistemic model extension. The general idea is based on [23] but extended to work in multi-agent scenarios, modeling each entity with knowledge and cooperation skills through time, using epistemic and temporal reasoning modalities in combination [70]. The system is modeled using the ISPL language where the faulty behaviour is automatically injected using a model-intrusive separate agent. Their research activity targets network protocols in particular, aiming at the study of their knowledge robustness and recoverability under bit/packet losses.

Only recently in [51] they presented a work on fault tolerance of an autonomous vehicle subject to degraded underwater environment conditions. The different parts of the embedded system are modeled as single epistemic agents and endowed with temporal reasoning skills. The objective is ensuring a correct cooperative self-diagnosis of the parts in case of fault, and an admissible response to such faults. In this work, following in the footsteps of [23], they propose their method as an engineering utility to prepare the system with the intended requirements and correct model behaviour.
To do this they first translate a Matlab/SimuLink model of the vehicle to ISPL language (by discrete abstraction of its continuous behaviour), inject faults automatically from a library (a table of possible faults) and model-check the resulting formal model against system requirements with the MCMAS model-checker. In case of a negative answer to the model-checking problem, MCMAS returns a counter-example that is then translated back to the Matlab/SimuLink framework with a nominal inverse procedure, in order to assist system engineers to understand the fallacies of the model in a simple and familiar form. This is seen as a refinement step that goes on until the specification is positive to model-checking.

Although interesting and effective, this approach has limitations. To begin with, the underlying temporal reasoning engine is based on CTL and, as such, it does not have native means of reasoning about the dynamic behaviour of an hybrid system such as the underwater vehicle they consider. The authors rely on a non better specified notion of discrete abstraction for that, plus its reverse procedure, with no mention to preservation or loss of language expressiveness in either. Moreover, a satisfactory comparison between using their dependability assessment techniques in embedded systems domains with and without the epistemic modalities is not present, whence one can wonder whether the same results could be achieved exclusively with temporal operators (and at what ease).

We will see this logic-based fault injection further developed later on in the chapter. For now let us just recap and give structure to what we have seen so far, introducing - by distinguishing - three types of Fault Injection.

1.2 The three types of Fault Injection

At this point we show the relation of Fault Injection techniques to system dependability, separating practices by their level of abstraction. As mentioned above, we will allow ourselves to distinguish three types of Fault Injection techniques, all carrying the same name but none being functionally related to the other application-wise.

Injection of the first type

Following the lines drawn at the beginning of section 1.1, the first type of injection concerns hardware faults and consists of testing dependability of hardware in system components. The purpose of this type of fault injection is that of seeing hardware responses to physical uncontrollable events such as thermal aging, electromagnetic radiation, etc. This type of injection is performed by either hardware manufacturers or software companies wanting this kind of third party faults to not compromise their code, limited to the hardware manipulations they need to rely on. Importantly it is an injection activity that is undertaken just before deployment.
The results of the analysis highly depend on the hardware and architecture of the system.

Injection of the second type

A different story goes for the injection of faults aimed at software bug tolerance. This type of Fault Injection, rather called *bug injection*, aims at seeing how software responses to software bugs and how catastrophic their effect can be at worst, highlighting fragile parts of software code and have a more or less general feeling about software robustness. As an activity, it is taken just before deployment and the results depend exclusively on software logic. Traditionally this type of fault injection has been applied to complex end-user software, although lately also focused on interface interactions between software components [75][3].

Recent work in the area [64] has highlighted common deficiencies of fault injection approaches as interface interactions are exploited for the injection of faults. In particular the authors show that obtaining representativeness at the interface level is not straight-forward and needs to undergo a handful of care measures. For example they show that buggy code of a library is most likely to corrupt several bits at once rather than only one as it is usually assumed in the literature. In addition they show that control-flow errors (i.e. errors due to unexpected executions of code in terms of instructions) are most often statistically correlated with data-errors in main memory. The gist of their work, as long as it concerns our interests, suggests that it may not be trivial to inject faults at the software interface level if a lot of care fails to be put in the process. Moreover, if we were to augment their results to be more faithful in how errors' side effects propagate in a general computing architecture, we could introduce registers, caches and parallelism in the picture, and if we further were to vary the code compiler and the machine where the code is run it is not hard to imagine how far spread the propagation of errors would go.

For software certification of safety-critical systems, where the architectural properties and code evolution model are not supposed to change much at deployment time, the authors implement the tool SAFE [41], that uses the considerations in [64] and similar to inject only representative faults in the interface code of the system components. This is done automatically after an appropriate selection of fault types from a given fault library by system engineers. At the end of the procedure, SAFE provides summary statistics of the system robustness, obtained by non-intrusive monitoring of the code runs, in terms of user-defined predicates over the system traces.

Injection of the third type

The third type of fault injection is *model extension*, where the whole system is considered in its design-modeling abstration and augmented with faulty behaviours there. The works published in [23] and [49] belong to this category. The fault injection and dependability assessment of the system is performed parallel to engineering and grounded to support the notion of refinement at every step of development. Importantly, the results of the analysis have connections to hardware, software and implementation details to the only extent to which the abstraction is concerned and no more. In the common case this boils down to assuming, effectively, only behavioral connections between the model and the actual system implementation.

The peril of mixing types

By the very reason that all three types of injection go under the same name in the scientific production, trickiness may arise. A possible source of confusion involves Software Fault Injection, because not always it gets explicit whether SIHFT or *bug injection* or other forms of injection that make use of software grounds, are intended to be.

An example of that kind of misconception is present in [66], where the authors aspire to assess SHIFT effectiveness in a symbolic way, moved by the claimed lack of coverage of standard SHIFT techniques. This idea is flawed.
The whole point of assessing system dependability using SIHFT, as we saw in section 1.1, is to avoid to change bits and memory values directly by hardware, in order to protect its physical integrity. Still, in SIHFT, it is hardware where the stress is to be placed and the program semantics needs to be kept very separate. As opposed to that, in [66] the authors rely, mistakenly, on a logical representation of the computation model, not on its actual implementation on the machine as it is claimed there.

Fault Injection for Model-Based Design

From the main three classes of fault injection reported in this section, others branch. In a recent paper [87] the problem of perturbing model-based simulation runs in Matlab/SimuLink is tackled; we may call this approach *Perturbation-Based Fault Injection*. The purpose is to check, although in a non-complete way, that always small perturbations at the input interface level of system components lead to small perturbations to the output. To do this the authors superpose (i.e. inject) the interface signals with artificial spikes, shifts and white noise in a systematic way, up to a fixed tolerance threshold, to see how the system behaves in response. They develop a computationally feasible algorithm for that injection and propose geometrical optimizations based on convex representations of the faulty signals.

They propose their work to be used at the early phases of system design, to identify lacks of robustness of system models since the early beginning of its construction, directly in the well-established environment of Matlab/SimuLink. The idea, although certainly novel, does not exploit the typical modularity aspect of Matlab/SimuLink models, yielding to a huge number of states to be checked, by simulation, for even a small set of interface signals. For the same reasons this approach cannot afford re-usability, which is of clear key importance in standard model-based design practice.

Notice that in this work, the injection obtains on the model-based system simulation and, as such, it eludes the categorization in either of the three types of Fault Injection outlined earlier in the section. Likewise the literature presents works that exploit fault injection in some other unexpected ways. We focus on the outlined three types only, because those only are needed to understand for our concern to understand the birth and evolution of *model extension*.

1.3 Contract Based Design & Fault Injection: CBSA

An interesting prominent example of fault injective model extension, as a derived evolution of [23], is covered in [20], where contract-based design (CBD) is exploited as a development basis of the system and the fault injection is put forward at each phase of refinement. This approach, referred to as Contract Based Safety Analysis (CBSA), stems from the need of:

(1) supporting a top-down and automatic fault-tree construction starting from the definition of a Top Level Event (TLE) and

(2) understanding its logical causes in terms of contracts' basic interactions.

The approach consists of taking a correct hierarchical contract-based tree-like architecture (the model) of a system and augmenting the top-level contract specification with two boolean ports f_1 and f_2 that are to represent, respectively, the failure of the assumption and the breaking of the guarantee (the extension). Given the top-level contract[1] $C = (A, G)$ they define the new, extended top-level contract $C^X = (A^X, G^X)$ setting $A^X \equiv (\neg f_1 \rightarrow A)$ and $G^X \equiv (\neg f_2 \rightarrow G)$.
The informal semantics of f_1 and f_2 is of disengaging the expected component assumptions or guarantees to observe their side-effects in the model. Interface ports of the sub-contracts are automatically inferred, later, from the refinement specifications in a provably sound refinement-preserving way. After that is all set up, the fault tree can be built as a refinement process, starting from the top-level event down to the leaf-refinements.

The strength of the approach lays on the contract-based policy, that allows the embedding of the fault injective model extension at every stage of the development process, pairing up the fault-tree construction, step by step, with contract refinement. This possibly gives insights about problems and weaknesses of the system during design. Moreover, due to its hierarchical, modular architecture, scalability and re-usability is easier to achieve than using traditional methods.

In the paper the comparison with the techniques in [23] is also discussed. Traditional techniques tend to construct single fault trees for each component imple-

[1]In [20], contracts are pairs composed of of sets of traces, represented symbolically as LTL formulae. We will offer a thorough presentation of contracts and their use in engineering in chapter 3.

mentation (which is typically trivial) and present the final tree as a disjunction of those. As a result, a *flat* two-level tree is obtained with no much information about structural dependencies of faults and poor scalability in general. With the contract-based framework this all disappears and the fault tree gets to be constructed incrementally assisting the system construction.

Despite its prospective usability, the proposed approach in [20] is although still not ready for a complete utilization. First of all it has to be said that the contract-based engine lying below the model extension is built on OCRA [39], which supports the temporal logic of HRETL [38] natively, sugared by the OTHELLO syntax [39]. It would not be surprise to see an extension of [20] towards the support of that language expressiveness in the near future. Moreover, as suggested by the very authors at the conclusion of the paper, the faults can be themselves specialized from mere boolean ports to full temporal formulae - although this passage does not not look trivial from the automatic injection point of view - in order to spot what truly is critical for the system. Strengthening the language expressiveness as mentioned not far earlier would also be of help for that same aim.

1.4 Summary of the chapter

We described in the present chapter common conceptions and trends of Fault Injection technologies. From fault tolerance of hardware (*Hardware Fault Injection*), its evolution is depicted towards the dependability assessment of whole complex software systems (*Software Fault Injection*, or *Bug Fault Injection*), either seen as OS-based colossi or interactive embedded parts. Different types of injections are described for different needs and objectives, intentionally driving our discussion converge towards *Model Fault Injection* (or *Model Extension*) and the contract-based fault injective CBSA, as an interesting in-progress proposal.

Importantly, we saw that tracking how faults get to propagate in software system implementations is hardly attainable in general and a model-based approach offers a good alternative to that by looking, from a logical perspective, at what can go wrong at the component interface level, where components are specified by assumption-guarantee gadgets (contracts) whose actual cause of failure is no more relevant than the fact itself that either of contract elements gets broken.

Chapter 2

SOA modeling and error management

Service-Oriented Architectures (SOAs) are getting growing attention by the system engineering community because of their ability to provide customizable ways to make systems interact and build well-defined complex systems. Lately also Cyber-Physical Systems are adopting this paradigm and formal methods are thus needed as a means for ensuring their correct working according to design expectations. Formalisms have been proposed to tackle the problem of formally secure their behavior but no work is present at the best of our knowledge on the rigorous analysis of their faulty behavior in terms of Fault Tree Analysis (FTA), Failure Mode and Effects Analysis (FMEA), etc.

Here a brief overview on the analysis techniques of such systems is proposed, with particular attention to their safety aspects and differences from the traditional paradigm of regarding such systems. We present a network-based fault injection technique that has been applied to the field first, then we overview published techniques to model Cyber-Physical, possibly SOA based, Systems. We present them according to the purpose of their conception, stressing the innovative aspects whenever possible.

Importantly, here we are not targeting the survey of modeling techniques of Service-Oriented Architectures from their design point of view: this subject will be later minded within the matters of chapter 5. Instead we will talk about existing abstractions and different options to highlight dependability challenges and put the problem that we are about to address into context.

There have been countless attempts to model SOA systems in the literature, at every level of abstraction and every domain where those techniques could find application. All of these works try to grasp fundamental architectural aspects targeted at the specific domain where the authors intended to put their own interest. This chapter elucidates on this topic with respect to our needs, presenting relevant works related to ours. Considerations are given in the final part of the chapter, were we outline a list of common desirable properties that languages for the modeling of

SOA, we expect, should have.

2.1 Injecting Faults, classically

When new implementation paradigms are encountered in software engineering and in industry especially, it is common and recurrent practice to submit the relative products to the V&V unit, that has to figure out ways to assess their correct working, pursuant to design expectancy. This is done through testing on the one side, and formal methods on the other. In a similar fashion the story replicates when the V&V unit confronts system dependability. This is treated, primarily, through hardware and software fault injection and through model extension only at a later stage, once automatic tools are available. The latter option is still not as popular at the industry level, whereas hardware and software fault injection is seen as part of ordinary testing activities.

The material on model extension is thus limited, in the field of SOAs, where even the paradigm itself is not solidly established over the cyber-physical domain. This section only considers classical injection of faults, without further digression on model extension.

There is a consistent number of works on the injection of faults in SOA-based Cyber-Physical Systems, among which we decided to mention only [73][71][1]. From there we also took inspiration for one of our use cases of chapter 5, the Thermostat example, that we slightly modified and enriched. Despite not being classical fault injection the focus of our study, it is at least fair to mention the explorations faced by the cited work and differences with respect to ours.

The approach adopted there is based on injecting faults at the network level — when messages are already in place to be encoded and sent through — by perturbing the Remote Procedure Call (RPC) parameters within the communication protocol at the middleware layer. This differs from other approaches in which faults are injected at the code level, which act on the construction of messages within the service software procedure. A tool, WS-FIT for Web Services – Fault Injection Technology, is presented. With the intent of focusing on the exchanged messages instead than on the modification of code they show how comparable performances can be obtained to code-based approaches.

Their fault model is thus focused on communication between services in a network. Their dependability is bound to be resilient to crash of services (either client or server side), hang of services, corruption of data in middleware, duplication, omission or delay of messages [72]. Protocol messages are parsed, injected by user-defined scripts and re-emitted to the network in protocol format. As shown in the paper, this procedure is overall effective.

As a foundational point for their work, the authors provide a list of SOA related

[1]We refer to chapter 1 to recall the possible alternatives to this approach, that can be adapted to the context of SOA by considering them as large software systems.

QoS requirements (i.e. desirable non-functional properties), that we report in the following and discuss:

– *Availability*: are services always reachable?
– *Accessibility*: does the system reply to accepted requests or does it hang?
– *Performance*: does the system perform well time-wise, in terms of internal latency of servicing a request and throughput (mean fraction of time to serve a request)?
– *Reliability*: to what extent can the system cope with internal/external faults?
– *Security*: can services be trusted?
– *Integrity*: is the state of the system always consistent with its expected behaviour?
– *Regulatory*: is the system compliant to specifications and protocol standards?

Noticeably they do not include jitter, but that is probably already part of the throughput in this view. Now we will move right to the analysis of each item, to understand which ones we will be able to handle, which ones to abstract and which ones to discharge. This will be, we claim, an adequate analysis on QoS for SOA.

Among those outlined ones, the authors of the work select only the first four to conduct their evaluation, most likely because it is problematical to define suitable measures compatible with all seven. In the model-based view of the system presented in the second part of the thesis, we will consider only reliability and performance. This is due to the level of abstraction with which properties will be defined.

Regulatory has to be statically checked when the service is provided, non compliance being synonym of absence. Security is normally delivered using apposite security protocols whereas integrity is often ensured using checksums or similar methods at the middleware layer, so that they lay beyond our modeling concerns. Finally both accessibility and availability can be subsumed by a single bit indicating presence/absence of the required service, thus abstracting away the actual details from the models. Reliability is clearly at the core of our study and we will have all the structural means for its handling. Performance will need a good host of care, but it will nonetheless find its place in our contract-based system design picture.

For this latter issue notice the difference between classical fault injection and the model-based approach: where the former can exercise the actual system by simply adding no-operation cycles between receiving one message and sending its response (as it is proposed in the article for the internal latencies) we will more clumsily need to model time explicitly. This is the price to pay for abstraction; nevertheless its treatment will find the greatest elegance — as already mentioned — by leveraging on contracts.

By injecting faults at the network level instead than on the software implementation of the protocol, the presented paper emphasizes the need of rethinking the way of how fault injection should be conceived, upgrade to a more abstract tier, with less complications.

Yet the fault injection of [72] is not logic-based (in the sense that we defined in chapter 1): it is still on the side of the V&V process where the system is tested, in phase of implementation. In the next sections we will make a move towards the other side of the V&V process, focusing on a more principled logical treatment of Service-Oriented Architectures and general Cyber-Physical Systems (i.e. not necessarily SOA-based CPSs) suitable for their early understanding in design and, ultimately, for their verification.

2.2 Modeling of SOA and relative faults

In this section we report modeling criteria that have been devised in the past for the modeling of SOAs and their faults, with the explicit intention to get rid of implementation details and make their logical properties explicit, instead. There is no aim, except for the last work of section 2.2.2, towards the verification of requirement over SOA systems for the works presented in this section: the only purpose is on understanding what can be ameliorated in the design by considering a purer view of the architectural meta-model and identify issues that may arise in the topological architecture.

2.2.1 First attempts to formalize services

In [40] the authors provide a simple formal fault model for web-based SOA by abstracting all the implementation details away, confined to no architecture or network topology.

From the perspective of users of web services, they define a fault with generality as a triplet $\langle \mathbf{al}, \mathbf{tp}, \mathbf{ssp} \rangle$, which represents:

1. The **<u>a</u>rchitectural <u>l</u>evel** where the fault occurs, which is a phase among *Service Discovery Infrastructure*, *Server-side End Point*, *Client-side End Point* and *Service Provision Infrastructure*.
2. The **<u>t</u>ime <u>p</u>hase** when the fault occurs, which enumerates in *Infrastructure Discovery*, Client or Service *Registration*, *System Configuration* and others.
3. The **<u>s</u>pecific <u>s</u>ervice <u>p</u>arameters**, which can be any, tracking the faulty/healthy condition of the system in a higher or lower abstraction level (e.g. this can be a binary variable Faulty/NotFaulty or a more complicated status comprised of a tuple of multiple variables with timestamps)

For example, a session failure might be represented user-side as the tuple

$$\langle \texttt{Service\_Provision}, \texttt{Service\_Delivery}, \texttt{Faulty} \rangle$$

With this conceptually simple fault model available for one single fault of one single user, one can define more elaborate faulty configurations for each user as a tuple of these, then a tuple of faulty configurations as an overall fault model for all users. If the fault model is judged too poor, we can refer to the taxonomy of [27] for a larger, more comprehensive classification of faults, that can be modeled by similar means.

Therefore, the proposed model is general enough to leave different levels of expressiveness to the modeler, possibly allowing both functional and non-functional requirements to be expressed. Nevertheless it is easily understood that the model is too trivial and does not provide efficient means to deal with the state-explosion, which is granted under the given conditions. A more suitable approach — both in terms of modularity, scalability and inclination to undergo verification — would be given by a contract-based design of the system, injecting faults as controlled failures of contract assumptions and guarantees. This hints at a possible revisiting of the CBSA methodology presented in section 1.3 in terms of services and service faults. This is left uninvestigated in this thesis and deferred to future work (see section 6.3.3).

Closer to the concept of contract, the work in [93] aims at formalizing the overall concept of SOA through service behaviours. A service implementation is seen as a pair $\langle P, N \rangle$, where P is a set of processes (or rather functions), identified by their programmatic interface and N is a relational group of edges that link the processes in P w.r.t. functionality and interdependability. A Service-Oriented Environment is then defined as a time-variable list of such service implementation pairs.

The formal apparatus of the paper is quite poor, however it puts forward catchy hints that strengthen a purely functional view of the SOA set-up; one that neglects the need for implementation details in the modeling and keeps the network structure away from single service's functionality. In this regards, they further stress how important it is for such models of SOA to capture the context-invariability at its very basis: services must behave independently from the application they are employed in. As we will see in chapter 3, this view fits well in the paradigm of contract-based design.

2.2.2 A complete formalization of services

Contemporary to the developments of [40] and [93], another line of investigation started out in [63][62], to model Service-Oriented Systems, ending up in the complete formalization of [26]. In this work services — *Rich Services* according to the authors' terminology — are viewed as functional relations between input and output streams, where streams are intended as sequences of messages. Streams are the authors' way to represent behaviour of services at the interface level, over time.

Streams have clearly a natural correspondence to temporal traces, often encountered in the formal verification vocabulary. In this sense input/output streams can be redefined as contracts (see chapter 3). Moreover, they define a notion of *composition* and *behavioral refinement* that is similar to that of parallel composition and refinement for contracts, in fact reconcilable to that. Importantly, services are defined as partial functions and explicitly they are intended to work under a restricted subset of all possible conditions. Components on the other hand are supposed to attach a behaviour for all possible inputs. We will come back to this point in chapter 3.

In a follow-up unpublished paper, [61], services are reconsidered from the same set-up, this time aiming at their architectural interactions in SOAs, with attention to dynamic binding, functional variability and system hierarchy. Rich Services are composed of several sub-services at the application layer. In turn, sub-services can themselves be Rich Services and connect with each other using an internal bus, that is also in charge of communicating with the outside through an input/output interface. The bus is seen in this work as a message orchestrator. Optionally, the bus has available a number of utilities at the infrastructural layer, such as encryption/decryption of message, logging, service policies and others. These are bound to the regulation of the functional behaviour of the contract towards the outside world[2].

From the logical perspective, Rich Services are modeled by three modules, that separate roles. The first is the functional module. The functional module, given by the behavioural composition of the sub-services' logical descriptions, never directly communicates with the outside. Before, it has to go under the supervision of the interface module — given by the behavioural composition of the bus' and infrastructural utilities' logical descriptions — which regulates the inputs and outputs of the Rich Service in order to manage compliance with the functional module. Finally there is an interface factory module that, based on the current output function of the Rich Service and environmental information, provides new interface modules for the Rich Service to expose. The construction of this latter module is arbitrary and depends to how the service functionalities change over time.

The value of the works presented here lays on the separation of the topological structure of the SOA from the functional aspect of the single sub-components. In other words, the proposed apparatus shows how functional entities transform to services and play roles according to how the architecture itself is laid out. In this picture, the functional module is an abstraction of the service discovery and execution, whereas the interface module represents the binding of the service to the architecture. Notice that the binding does not happen with another service as in the common understanding of SOA, but with the whole rest of the architecture. This is also consistent with our previously professed context-independence, that demanded services to be free from constraints imposed by the outside world — such as the embedding structure. Finally the interface factory module represents modifications in topology, indicating to the interface module, for each time instants, what the functionality module should expose to the environment and what inputs it should expect to be given by it.

2.2.3 A model extension of the service formalism

The service formalism of [26] drove the authors in another direction, presented in [47]. This article, where failure management is discussed in the SOA domain, is probably the closest to our research among those from the same authors, presented in this section. In addition to the previously described features, this work enriches the picture with service failures, either in the form of self collapse or due to communication issues — such as unavailability — between many. Faults are injected according to user specifications in textual language format and a verification procedure is set forth. Since we will operate under similar circumstances, it is valuable to

[2]This architecture has been originally proposed in [6].

make a digression on the intended methodology and anticipate the different aspects with respect to the handling of the matter subject of the second part of the thesis.

The paper models *architectures* and *configurations*. Architectures are composed of *roles*, *channels* and *messages* that define how the services interact, as partial functions defined by Message Sequence Charts (MSC), on the network. Configurations, on the other hand, contain *components*, *connections* and *signals*, that implement in a 1:1 correspondence roles, channels and messages of the architecture. This specification is given in SADL (Service Architecture Definition Language), a language developed in this very work for the textual specification of services. SADL assimilates faults directly in the model definition, with the possibility of attaching to each component a can_fail flag. Moreover it has special constructs for failure detection and mitigation that specify under which conditions the detector acknowledges the fault and how is the reaction.

For once, the modeling of the faults is not an end in itself but is followed by a verification support to the methodology. Following the translation procedure outlined in [48], the Message Sequence Charts specifying the interactions between roles are translated into state machines and then in Promela language, so to enable the verification of temporal properties by means of the SPIN model checker. The state machines representing the nominal behaviour are linked to the fault detection and mitigation machines by non-deterministic transitions, triggered by a dedicate failure injector process that sends failure messages, non-deterministically, on the channels. Those messages will affect only those machines that can_fail. At this point the verification process can take place, checking for example, whether under the action of a given number of faults, desirable behaviours of the nominal system eventually happen regardless of the presence of those faults.

The approach that we will present is based on model extension and will keep network topology and single components separated. However, unlike [47], we will allow for a concept of incremental refinement, based on design contracts, that will let us specify the level of complexity of each component at arbitrary depth. The proposal in [47], on the contrary, is solely based on the interactions between roles, implemented as MSCs. The internal dynamics of the single components is thus totally denied.

We will not only study the behaviour of the system under failure hypothesis of some of the services; instead we will move for the option of describing the configuration under which a service may be brought to failure, using automatically generated fault trees. And the separation between network and the single components will not be only founded on ease in design specification, but exploited in the contract definition, even for the specification of temporal constraints. Moreover, having NuSMV as a back-end, we have all the positive prospects to end up in a complete tool for the system stepwise design and verification, that may eventually find application even in industry, beyond pure academic domains.

2.3 Property Verification of Cyber-Physical Systems

In this section we consider the modeling techniques that exist in the literature to model SOAs and generic Cyber-Physical Systems, fostered by property verification concerns. There are three mainstream approaches available in the literature for this kind of support: process algebras, petri nets and automata-based. An instructive survey of these methods is presented in [10], where particular attention is put on securing services at the stage of composition, when a server needs to combine, or rather orchestrate, two or more interactions patterns of the slave services to deliver the required answer to some user's query.

Such compositional demands are not at the focus of our research, even though we will present a novel technique, in section 6.3, that is able to handle the close concept of dynamic system reconfiguration. However, in view of the fact that SOAs have a broad literature dedicated to web-service interaction patterns, we owe some words to how those are modeled. This is not worthless to our ends, since other approaches adapt the same existing formalisms to the CPSs' scene. Here we limit ourselves to a representative subset of those, a few noteworthy instances of each of the three methods, stressing on the elements relevant to our work.

2.3.1 The Time-Space π-calculus

Process algebras are mathematically-founded symbol manipulation systems that describe processes and their intertwining in concurrent computing ambients. They abstract the computations using simple interaction primitives whose only awareness is on the exchange of symbolic information between independent sequentially executing processes, that is usually conceived as happening on *channels*.

Among process algebras, the most popular is most surely the π-calculus which, together with its several variants, has been widely used in service applications for its ability to create channels anew and transmit them to other processes. This becomes important when interactions such as service discovery, that sees the user asking the service broker a new channel to bind to, require modeling. Moreover, process algebras come with a transitive relation, representative of system execution, whose closure encompasses all the admitted computational behaviours of the described process. The *only* big nuisance is that membership to the transitive closure is not decidable for most interesting process algebras, so that property-verification is usually faced with either unsound or incomplete action modalities.

Related to our work, [90] describes Cyber-Physical Systems using the *Time-Space π-calculus*, which extends the expressive power of π-calculus with time, energy and space operators. For example, when guarded by the time-interval operator $\mathsf{Int}(\mathsf{t}, \Delta\mathsf{t})$, a process is permitted to start only within the specified time window $[t, \Delta t]$. Time engages in discrete steps in this work. The architecture of the system is formalized as a group of interacting sub-systems (i.e. the architectural services), regarded as processes in the language. After that, their effective parallel execution is formally checked against deadlocks and live-locks, or otherwise against the reachability of a given configuration.

There is no mention in this work about the possibility of injecting faults in the model. Injection could be attained by composing the system with suitable injection processes, using the choice semantics to compromise processes upon messages acceptance, by disabling non-deterministically service operation (e.g. transforming processes in the form $a(x).P.0$ into $a(x).(P.0 + 0)$. Yet this procedure would be shallow, because nothing could be said about the internal system dynamics of the service implementing components.

In general the risk in using process algebras to realize system dependability is misuse. Process algebras are used to prove things about highly abstracted systems. They are preferred to other approaches when the core interest is in communications and concurrency, whereas we are on a ground where single services and their failures cry for the spotlight. Moreover, having to deal with the Turing-complete expressiveness of π, we cannot hope to solve all semantic requirements by means of automatic inference. This said, possible developments of π-calculus in this direction are not to be excluded, at least for academic interest.

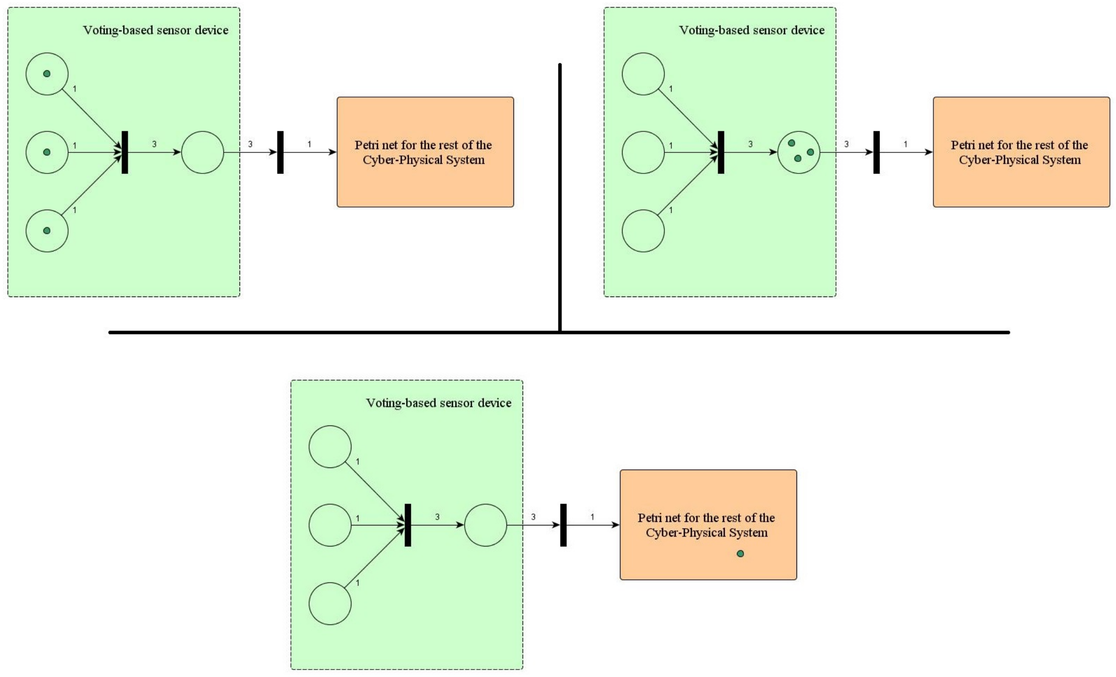
2.3.2 Labeled Hybrid Petri Nets

Petri nets are graphical representation of reaction processes, originally of chemical type, that flow units of interest, so called *tokens*, through *transition blocks* and between *places*. Formally they are bipartite directed graphs that evolve non-deterministically by circulating tokens from one place to the other when the separating transition block is triggered; only one transition of many can fire at each step of evolution and, when it does, this happens instantaneously. Arcs connecting the parts of the bipartite graph have weights, that indicate the enabling number of tokens and their consumption for the transition upon triggering.

Next to being employed for the description of biological systems, Petri nets were reconsidered for the modeling of computer concurrency systems, where transitions were rethought as algorithmic transitions between control points (program states) and tokens as representative for the number of processes reaching a common program state.

For a contextualized example, if the interest was on modeling a fault-resilient voting-based sensor device composed of three voting sub-sensors in a Cyber-Physical System, then one could construct a Petri net having three places, whose coordination possibly reaches the rest of the system only if all of the three sub-sensors make their vote available, as per figure 2.1. Notice that the total number of tokens varies according to arc weights; this view is consistent with the initial modeling pertaining chemical reactions. In the case of the fault-resilient sensor, once the transition reaches the outer CPS there is no longer the need of keeping three separate measures.

As figure 2.1 reveals, the plain Petri network as-is is too poor for describing the complexity of Cyber-Physical Systems. Variants have thereby been proposed in the literature to extend some of Petri net functionality, seeing the birth of Timed Petri



Voting-based sensor device
1
1
1
3
3
1
Petri net for the rest of the Cyber-Physical System
Voting-based sensor device
1
1
1
3
3
1
Petri net for the rest of the Cyber-Physical System
Voting-based sensor device
1
1
1
3
3
1
Petri net for the rest of the Cyber-Physical System

Figure 2.1: Petri net evolution for a fault-resilient voting-based sensor device. Circles represent *places*, black marbles *tokens*, black rectangles *transition blocks*.

Nets (TPN — having time-triggered transitions), Stochastic Petri Nets (SPN — where transitions are regulated by probability distributions) and Hybrid Petri Nets (HPN — whose tokens are fractional and move according to the laws attached to the transition arcs). For all the interesting purposes of our work, all variants of Petri nets can be somehow reestablished using state machine semantics[3].

In [86] the Labeled Hybrid Petri Net (LHPN) formalism is proposed, at the edge between classical Petri Nets and Hybrid Automata, to represent and verify Cyber-Physical Systems. Driven by the need of localizing the continuous part of the system dynamics, still maintaining the graphical expressiveness of Petri nets, the formal specification combines the two formalisms obtaining, as a result, Petri nets that evolve continuously within places (limited to first-order differential equations with constant coefficients — i.e. exponential decays/uprisings) and go through transitions according to guards and reassignment of variables. Failure transitions can be specified as part of the model as transitions leading to dead-ends. Verification will take care of checking whether the reachability of the system contains such undesired failure transitions.

[3]Interestingly, even the formal techniques for the verification of Petri nets are often inspired or connected to automata techniques, including region and state-space partition and abstraction.

Verification of general LHPN is non-decidable, therefore sound and complete semantic checks are not applicable in this framework[4]. In the presented approach, the state explosion problem is mitigated by semantics preserving graph transformations and forgetful abstraction, partly based on convex inclusion.

Petri nets are deemed more intuitive than automata at modeling, because they make the transitions between one place and the other explicit on arcs. This is part of the motivation for the work in [86]. However the non-expert user might find them anyway hard to master, because they are not standard.

In chapter 5 we will indicate derivatives of the Unified Modeling Language (UML) for the representation of complex interactions under the SOA paradigm. Speaking of this, the approach based on Petri nets might not be the most appropriate, because services have an intrinsic notion of communication, that lacks in Petri nets. Also, their structure does not efficiently support hierarchical design, that is paramount for scaling. Our methodology, founded on contracts, will be shown to support that and more.

Finally, while presenting a study case with a fault-tolerant sensor (similar to that of figure 2.1 in fact), the paper provides no mention to the dependability analysis of the systems in terms of standard techniques such as FTA or FMEA. Our interest will converge there because of our motivation lays on industrial needs.

2.3.3 Networked Event-Data Automata

Automata, born as representational devices for the analysis of formal languages, have become among the most popular formal ways of modeling independently evolving system components and their event-based interactions. Noteworthy examples are Input/Output Automata and Timed/Hybrid Automata.

Automata are mathematical structures characterized by a set of states with internal actions, a transition relation, an alphabet of visible actions occurring on transitions and possible markers for the input and final state/states. Variants exist of different sorts. For example timed automata supplement the structure with time, modeled as clocks, whereas hybrid automata generalize system evolution accounting for both discrete and continuous dynamics, with transition guards enabling transitions and state invariants forcing state departure in case of unsatisfaction. We assume familiarity to this concepts by the reader.

Here we decided to put our attention on one specific work, that applies model extension on networks of special hybrid automata called Event-Data Automata (EDA) [19]. Each EDA component is an independently evolving hybrid machine composed of *modes* (as opposed to the most common reference to *locations*) and transitions, each with triggering events, guards and effects on internal and output variables. State invariants, on modes, are boolean arithmetical expressions defined on the linear fragment. Flow equations, internal to modes, are limited to linear

[4]Reachability of LHPN is based on partitioning techniques of the infinite state-space into convex regions, represented as Difference Bound Matrices (DBM) [69]

differential equations with constant coefficients.

EDAs can ensemble into NEDAs, Networks of Event-Data Automata. Formally, a NEDA is structured by a set of EDAs, with an activation mapping α that takes each mode configuration to a subset of active EDAs under that mode (in other words α decides which EDAs evolve in function of a mode configuration). Semantically, all component EDAs evolve independently but synchronize with time. Internal transitions (i.e. event triggered change of modes) take no time.

In the example provided by the authors, an hybrid system switches between two batteries upon one coming to low energy. The *active* battery is either one, depending on the system mode being set to *primary* or *backup*. However, when one battery goes from *active* to *inactive*, it leaves the frontline and disappears: it does not interact with the other EDAs nor it evolves by its own. Upon recovery it is optional to restore the component state as it was before deactivation, otherwise it is restarted, memoryless of the prior activation.

Fault injection is accomplished on the nominal system description as model extension. A separate error model is provided for that aim, written in SLIM language, as an adversary automaton that interacts with the nominal model through an external interface. The external interface of the fault model is accessed through an injection of new events in the nominal behaviour, by adding the chance to trigger according to an exponential probability distribution. The nominal model and the fault extension run concurrently to encompass all their combined behaviours.

The entire framework is supported by the language SLIM (System-Level Integrated Modelling), an extension of (a subset of) AADL[1], supporting hybrid dynamics and faulty system behaviours. SLIM is eventually translated into SMV language and NuSMV used as a model-checking back-end. The Markov-Reward Model Checker (MRMC) is used for the verification of probabilities related to stochastic faults. Automatic dependability analysis techniques are supported, such as the already mentioned FTA and FMEA, with the supplement of probabilistic information, that are visualized by dedicated graphical viewers.

Extended AADL has been intentionally designed for supporting the analysis of complex distributed Cyber-Physical Systems with faults and the SLIM language is nothing but the resolution of isolating its relevant features in the direction of verification purposes constrained to fault injection. Understanding why it is worth of our mention thus shouldn't become too puzzling.

Yet, our approach diverges from [19], firstly at a conceptual level. For the reasons that we have been writing about for all the section through, our CPS conception is based on services and relative compositional architectures. Our focus is on description of modularity aspects grounded on contract-based design; hybrid modeling comes as a secondary concern. However we share one important characteristic with [19], that is the construction of fault trees: the tool used in the second part of the thesis, XSAP, is creation by some of the authors of the paper and thus uses closely related analysis techniques.

Works like this raise our confidence in the adoption of NuSMV derivatives/extensions for the analysis of Cyber-Physical Systems and put ourselves in believing that we will be able to benefit from future versions of the tool integrating features of shared interest.

2.4 Final considerations

Several techniques have been proposed in the past to assess the dependability of classical systems, but the research area about the dependability of complex Cyber-Physical System is still at its infancy and little work about the logical treatment of SOAs combined with Dependability Analysis can be found in the literature, to the best of our research activity.

Remarkably, from the literature we were able to identify some interesting common features that are at the very essence of SOAs or complex Cyber-Physical System in general. Although not being strictly necessary for their logical treatment, they are at least desirable[5]

- Cyber-Physical SOAs are heterogeneous in nature, and this heterogeneity should be supported by the modeling and verification language either by means of logical abstraction or specific composition of different modeling/verification paradigms (see below).
- The philosophy underlying SOA is that services are functional, context-independent, not unique (i.e. many sources can give the same service-functionality) and ought to interact at the interface level. For this a clear definition of composability has to be addressed by the modeling language and the verification tool, interface-wise.
- In many cases, services are black boxes to the verification engine and they define a network topology that changes through time. Therefore faults — including the possible disappearance (dually appearance) of services from the network — are unpredictable, besides on a trust basis. Capturing the system evolution dynamics and its relations to occurring faults becomes thus essential for the design of a dependable SOA.
- Cyber-Physical Systems in real life are normally complex and characterized by many components. More and more the future will prefer hierarchy and modularity over flat solutions, because existing problems, such as heterogeneity, are gradually getting defeated. Modularity and hierarchy are the main forces to employ in order to get the road paved for system scalability; it is thus desirable for modeling and verification languages to reflect them in their constructs and get algorithms tuned accordingly.

[5]The list that we present here is not a bare synthesis of the properties outlined in the previous sections, but an additional supplement and integration, the condensation of features coming from an extensive survey of the literature, with works that couldn't fit in the thesis for either scope or space.

- Quality-of-Service (Jitter, Latency, ...) is a non-functional but pervasive requirement for Cyber-Physical Systems. Unexpected latencies can even be causes for a system failure in real-time safety-critical systems. A modeling and/or verification language supporting, at least partly, QoS is thus desirable. Classical dependability analysis practices would have to be enriched with QoS-oriented basic events in these regards. Top Level Events would also need to allow QoS constraints in their definition.

As we will be able to acknowledge after the mathematical foundations of chapter 3, the contract-based paradigm fits well most of the points presented above. Still there are issues, such as the timing aspect, that will need to be tackled with atypical techniques. The trick will consist in leveraging the service orientation and stressing on context-independence, separating the functional and the architectural viewpoints, handling timings supported by the latter.

Our modeling language of choice will be as general as possible, based on derivatives of UML, to accommodate non-specific engineering practice. Our methodology will be, anyway, easily adaptable to more specific modeling languages with a richer set of features and support of service orientation. This will be further explored in section 5.1.

Before diving to the heart of our developments, we sketch out the approach that we use in this thesis to address system heterogeneity. One of the prevalent lines to that aim consists in using a bottom-up approach, combining domain specific simulation/verification results obtained by different tool in one overall framework. This view is taken and lead by Ptolemy [67], supporting different hierarchically integrable components based on actor-oriented heterogeneous models of computation. The formal semantics of Modal Models is presented as the refinement of state machines down to heterogeneous components; where simulation takes the foreground for assessing component's correctness.

Similar ideas are presented in [53] for the treatment of heterogeneous Systems-on-Chip (SoC) and in [78] for Heterogeneous Rich Components. Although provably valuable and effectual, they do not cover the two aspects of service orientation and automatic generation of dependability artifacts in which we are interested and thoroughly founded on.

As we saw throughout this chapter, models of service orientation abstract from the component's implementation and heterogeneity details more often than not, focusing rather on the service interactions at level of their interface. On the other hand Cyber-Physical Systems are in most of the cases described in full detail, consistently disregarding the interaction patterns in the name of functional correctness, as it is done in the heterogeneous modeling frameworks that we just mentioned above, in the past few lines.

We locate our approach in the middle of this, in a trade-off position that is the starting point and core of our scientific contribution. Top-down, the plan is to let designers specify the system's expected behaviour in the form a contract with its relative compositional refinements (cf. chapter 3) and, in parallel, define a network architecture to support and cover the SOA-specific requirements. Single implementations are meant to be provided compliant to services' contractual descriptions and analyzed against them in virtue of fault resilience, with assistance of fault tree constructions and FMEA tables. Notice, importantly, that our methodology focuses on the SOA system's dependability analysis and, in this sense, offers a complementary and integrative methodology to bottom-up simulation-based approaches, as opposed to a contrasting one. To enable this kind of dependability analysis we will need to rely on a single specialized tool, XSAP in our case, that defines its own input language that all components are required to adhere to.

Chapter 3

Contracts for System Design

In the past chapters we gave an introduction to fault injection and the way it has been lately adapted to model-based system modeling. We then went through the state-of-the-art techniques for the modeling and verification of SOA and interactive Cyber-Physical Systems, showing strengths, weaknesses and possible links between them, bringing a view of adaptation through service-oriented systems. In both cases we acknowledged that reasoning on interfaces is beneficial against complexity and heterogeneity, plus it aligns smoothly to the latest formalizations of services present in the literature.

In this chapter we will put through the recent theory of contracts for system design [13], which provides a thorough albeit not always intuitive mathematical layer for system design. We already informally introduced contracts in the introduction and recalled them here and there in the follow-up presentation. The formal presentation of this chapter will provide the conceptual basis to understand the importance of the contracts around service orientation. Our aim in this chapter will not be on specifying all the formalities and details, nor it will be to develop further mathematics on top of that already existing. Instead, we will be interested in providing an intuitive understanding of the framework and make some considerations on its applicability in our context.

The presentation will be intertwined by formal definitions and simple examples, in order to engage in a new viewpoint and sustain the reason that make it so strong as advocated. We will discuss the generality and wide applicability of the contract-based approach, as well as its novelty aspect as a tool for system design. After settling the basis of the framework together with its intended connotation we will have the means to weigh up our previously expressed confidence in this remarkable mathematical framework for system design.

3.1 An algebra of contracts

In this section we present the algebra defined over contracts, following an unconventional way of exposition, i.e. starting with the definition of the contract algebra and explore its applications through examples. This is in contrast with the more common strategy of justifying the algebra by inference from practical concerns[13][12][14].

Definition 1. Let us consider $\mathcal{U} \in \mathsf{Sets}$ a set universe. A contract is a pair (A, G), defining an $\underline{\text{A}}$ssumption and a $\underline{\text{G}}$uarantee, such that $A \subseteq \mathcal{U}$, $G \subseteq \mathcal{U}$ and $\overline{A} \subseteq G$.

We write $\mathcal{C}$ for the class of contracts. Conventional presentations of contracts omit the last condition and validate its later introduction as a form of saturation or normalization, distinguished based on whether it is imposed as a requirement or implied by other axioms. The last condition of definition 3 *defines* contracts as saturated already, which entails them being in normal (or canonical) form. We prefer to take saturation as given because it makes the treatment straighter and off technicalities. In doing so we put ourselves in condition of losing some expressiveness because, classically, complementation is not always computable in Sets. This will not make a difference for our introductory purposes.

Contracts can be satisfied by running instances, called *implementations*:

Definition 2. Given a contract $C = (A, G)$ and $M \subseteq \mathcal{U}$, we say M is an implementation of C or M satisfies C, written $M \models C$, iff

$$M \cap A \subseteq G$$

Proposition 1. *There exists a unique $\subseteq$-maximal implementation M_C for every contract $C = (A, G)$ and $M_C = (\overline{A} \cup G) = G$.*

Proof: *$M_C = (\overline{A} \cup G)$ is an implementation of C, clearly, because*

$$M_C \cap A = (\overline{A} \cup G) \cap A = G \cap A \subseteq G$$

Moreover it is maximal, because for any other implementation M we have $M \cap A \subseteq G$ and thus $M \subseteq \overline{A} \cup G = M_C$. ■

Obviously there also is a unique minimal implementation, namely $\emptyset$, which is implementation of every contract. So we have an escalation of subsequent implementations from $\emptyset$ to M_C, governed by set containment. Moreover, if we pick any of these implementations, say M, then it is maximal with respect to a specific contract, namely the contract $\widehat{C} = (\overline{M}, M)$.

To close up the commutative square between inclusions of contract's maximal implementations and contract satisfiability, we are interested in the relation between $\widehat{C}$ and C. From the general implementation $M \subseteq M_C = \overline{A} \cup G = G$ in the escalation, we instantly derive $\overline{M} \supseteq \overline{G} = A \cap \overline{G} = \overline{M_C}$, by contraposition. Now it can either be the case of $\overline{M} \subset A$ or $\overline{M} \supseteq A$. However, the former case is not feasible because otherwise, by another contraposition, $\overline{\overline{M}} = M \supset \overline{A}$, whereas we know from $M \subseteq M_C = \overline{A} \cup G$ that $M \subseteq \overline{A}$.

For our general implementation M we thus have a contract $(\overline{M}, M)$ which generates it maximally and such that both $M \subseteq G$ and $M \supseteq A$. This brings us to the concept of dominance.

Definition 3. Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ contracts. We say C_1 *dominates* C_2, written $C_1 \preceq C_2$, iff

$$A_1 \supseteq A_2 \quad \text{and} \quad G_1 \subseteq G_2$$

Proposition 2. *Dominance is a partial order.*

Proof: *Trivial from*

(1) $\forall C \in \mathcal{C}.\ (\ C \preceq C\ )$
(2) $\forall C_1, C_2 \in \mathcal{C}.\ (\ (C_1 \preceq C_2) \wedge (C_2 \preceq C_1) \rightarrow (C_1 = C_2)\ )$
(3) $\forall C_1, C_2, C3 \in \mathcal{C}.\ (\ (C_1 \preceq C_2) \wedge (C_2 \preceq C_3) \rightarrow (C_1 \preceq C_3)\ )$ ■

Most of the unfolding of the proof above reduces to the partiality property of set inclusion. Moreover from Sets being a distributive lattice it could also be shown that dominance induces a distributive lattice $\mathcal{C}, \preceq$ on the class of contracts. Even more specifically, induced by $\preceq$ contracts inherit a boolean algebra $(\mathcal{C}, \sqcap, \sqcup, \overline{[\bullet]}, \mathbf{0}, \mathbf{1})$ from Sets where, given generic $C, C_1, C_2 \in \mathcal{C}$:

$$\begin{aligned} C_1 \sqcap C_2 &= (A_1 \cup A_2, G_1 \cap G_2) \\ C_1 \sqcup C_2 &= (A_1 \cap A_2, G_1 \cup G_2) \\ \overline{[C]} &= (\overline{A}, \overline{G}) \\ \mathbf{0} &= (\mathcal{U}, \emptyset) \\ \mathbf{1} &= (\emptyset, \mathcal{U}) \end{aligned}$$

Notice that the assumption of having set complementation available is paramount to construct contract complementation and thus the derived boolean algebra.

Proposition 3. *Let M_{C_1} be the maximal implementation of contract $C_1 = (A_1, G_1)$ and let $C_2 = (A_2, G_2)$ be just another contract. Then*

$$C_1 \preceq C_2 \implies M_{C_1} \models C_2$$

Proof:

$$\begin{aligned} C_1 \preceq C_2 &\implies A_1 \supseteq A_2 \wedge G_1 \subseteq G_2 && \implies \\ &\implies M_{C_1} = \overline{A_1} \cup G_1 = G_1 \subseteq \overline{A_2} \cup G_2 && \implies \\ &\implies M_{C_1} \cap A_2 \subseteq G_2 && \implies \\ &\implies M_{C_1} \models C_2 \end{aligned}$$

■

It also follows that $M \models C_1 \wedge C_1 \preceq C_2 \implies M \models C_2$, which says that everything a dominating contract implements, the dominated contract implements too. In other words, the lower-set *below* an implementation M with respect to set inclusion is entirely implemented by any contract dominated by $C = (\overline{M}, M)$

Let us see an instantiation of the framework on a trivial example in order to fix the concepts treated so far:

Example 1

Let $\mathcal{U} = \mathbb{N}$ be the universe and $C_1 = (Even, Odd \cup \{10\})$, $C_2 = (\{4, 6, 12\}, \mathbb{N})$ be contracts.

Easily $C_1 \preceq C_2$. The maximal implementation of C_1 is $M_{C_1} = Odd \cup \{10\}$. Clearly, any subset of M_{C_1} is also an implementation of C_1 (e.g. $Odd \models C_1$). Moreover, by proposition 3, any subset of M_{C_1} is implementation of C_2 too. Notice that the conjunction of the two contracts, being it by definition the infimum with respect to $\preceq$, is $C_1 \sqcap C_2 = C_1$.

Conjunction will turn out to be the interesting operator in the following sections because it combines different contracts preserving implementations for as much as possible. This is a consequence of contract conjunction commuting with the intersection of maximal implementations, i.e. $M_{C_1 \sqcap C_2} = M_{C_1} \cap M_{C_1}$.

However, given two contracts, there is an additional operator designed to combine them, which lays out of the algebra. This is parallel composition.

Definition 4. Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts. We define the parallel composition operator, denoted with the symbol $\|$, as

$$C_1 \parallel C_2 = ((A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}, (G_1 \cap G_2))$$

Parallel composition is the central operation between contracts, it is what distinguishes this theory from one that straightforwardly exhibits a composite boolean algebra; it is also what makes sense out of assumptions and guarantees.

To fully understand parallel composition we will have to wait until next section. For the moment let us simply see what it does on contract's maximal implementations on a couple of simple examples.

Example 2

Let $\mathcal{U} = \{a, b, c, d\}$ be the universe and $C_1 = (\{a, b, c\}, \{c, d\})$, $C_2 = (\{a, b, d\}, \{a, c\})$ be contracts. Contract conjunction and parallel composition are obtained, respectively, as

$$C_\sqcap = C_1 \sqcap C_2 = (\{a, b, c, d\}, \{c\})$$
$$C_\| = C_1 \parallel C_2 = (\{a, b, d\}, \{c\})$$

Maximal implementations of those are thereby the same:

$$M_{C_\sqcap} = M_{C_\|} = \{c\}$$

Example 3

Let $\mathcal{U} = \{a, b, c, d\}$ be the universe and $C_1 = (\{a, b, c\}, \{a, b, c, d\})$, $C_2 = (\emptyset, \{a, b, c, d\})$ be contracts. Contract conjunction and parallel composition are obtained, respectively, as

$$C_\sqcap = C_1 \sqcap C_2 = (\{a,b,c\},\{a,b,c,d\})$$
$$C_\parallel = C_1 \parallel C_2 = (\emptyset,\{a,b,c,d\})$$

Maximal implementations of those are thereby the same:

$$M_{C_\sqcap} = M_{C_\parallel} = \{a,b,c,d\}$$

There is a trend of parallel composition of generating new contracts with less restrictive assumptions than conjunction, while sharing the guarantees and admitting, as a consequence, the same implementations. The second part is intuitive, because guarantees generate from the same combining operator, namely intersection. It is less clear whether having more permissive assumptions would be a logical consequence of the construction or just an accident. The following proposition provides a clear answer:

Proposition 4. *Let $C_1 = (A_1, G_1), C_2 = (A_2, G_2)$ be contracts. Their conjunction dominates their parallel composition, both having nonetheless the same set of implementations i.e.*

$$(C_1 \sqcap C_2 \preceq C_1 \parallel C_2) \quad \wedge \quad (M \models C_1 \sqcap C_2 \iff M \models C_1 \parallel C_2)$$

Proof: *To prove the first part of the proposition we only need to prove that $A_1 \cup A_2 \supseteq (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}$, because containment of the guarantees is ensured by equality. However, $(A_1 \cap A_2)$ is subset of $(A_1 \cup A_2)$, which reduces our problem to checking, only, that $A_1 \cup A_2 \supseteq \overline{G_1 \cap G_2} = \overline{G_1} \cup \overline{G_2}$. Since $\overline{A_1} \subseteq G_1$ and $\overline{A_2} \subseteq G_2$ by definition of contract, $A_1 \cup A_2 \supseteq \overline{G_1} \cup \overline{G_2}$ follows by contraposition. Moreover since $C_1 \sqcap C_2$ and $C_1 \parallel C_2$ provide the same guarantees, they have the same maximal implementation and thus overall the same implementations (by proposition 3).* ■

It follows from proposition 4 and definition of infimum that the result of parallel composition neither dominates C_1 nor C_2 in general, but provides a contract whose assumptions are generally stronger than at least one of the contract's. In other words, if either one of the contracts would have its implementation put individually in a context where the parallel contract applies, where assumptions possibly fail to match, it would not be liable for violating the guarantee and therefore it possibly would. The idea of parallel composition is to provide a contract admitting the shared implementations of the individual component contracts, although applicable in a relaxed context where guarantees of the one contribute to resolve the context of the other. From yet another viewpoint, parallel composition embeds the notion of two components individually unable to win the game of blending in with a context, although qualified to do that collaboratively.

Here is another example relating conjunction and parallel composition on an instance of the framework:

Example 4

Let $\mathcal{U} = \mathbb{N}$ be the universe and $C_1 = (Odd \cup \{2\}, Even \cup \{5\})$, $C_2 = (\{1,2,3,4\}, \mathbb{N})$

be contracts. Contract conjunction and parallel composition are obtained, respectively, as

$$C_\sqcap = C_1 \sqcap C_2 = (Odd \cup \{2,4\}, Even \cup \{5\})$$
$$C_\parallel = C_1 \parallel C_2 = (Odd \cup \{2 \quad \}, Even \cup \{5\})$$

Maximal implementations of those are thereby the same:

$$M_{C_\sqcap} = M_{C_\parallel} = Even \cup \{5\}$$

Again, contract conjunction admits all common implementations of the two contracts, whereas parallel composition is more open, admitting contexts that individual contracts can resolve only *together*. Together in the sense of considering, besides common assumptions, what the other contract is in position to guarantee, to embed collaboratively in contexts larger than what individually expected.

The substance of the framework is now starting to take shape but it is still legitimately blurry how this would make sense to a dependability analysis for Cyber-Physical components. This is going to be cleared up in the next section, once we take on a discussion about interface components.

3.2 Interface Components

The contract framework that we have seen so far is so general that it seems to have no relevance to the work on verification and dependability assessment that we will need using with services. To see how instead this is a wrong perception we will need to concretize the framework on application, introducing interface components.

Definition 5. Let $\mathcal{L}$ be a logical language over a set of variables $\mathcal{V}$ and a domain $\mathcal{D}$ and let the interpretation of constants and symbols form a boolean algebra. Then we call *interface component* an implementation of a contract whose assumptions and guarantees are expressed using the language $\mathcal{L}$. We call such contract an *interface contract.*

Interface contracts and components are nothing but symbolic ways to define contracts and their implementations. For example, using boolean predicate logic as a language over the domain of real numbers with standard interpretation of symbols and constants, we can express the contract for an interface component as a predicate formula over variables in the real domain.

The peculiarity of interface contracts is that they have a concept of language as part of their definition and the concept of variable as part of their language. In the specification of the standard theory of contracts is is customary to call variables *ports* and discriminate between visible versus hidden (or local) ports, between controlled and uncontrolled ports. We will only commit to the concept of ports, making no further distinction, at this level, between them.

Let us see an example of interface components defined using the language of predicate logic, showing how the theory that we thus far developed applies in this domain:

Example 5

Let $\mathcal{V} = \{x, y_1, y_2, z\}$ and $\mathcal{D} = \mathbb{R}$. Let $C_1^{x,y_1} = (x > 0, x > 0 \to y_1 > 0)$ be an interface contract. With standard interpretation this stands for the contract that works on environments where the port x is set greater than 0 and results in the environment having port y greater than 0. Ports y_2 and z are unconstrained.

The definition of C_1^{x,y_1} is defined over the universe $\mathcal{U} = \mathbb{R}^4$; it is a contract because the set corresponding to $\overline{(x > 0)}$ is included in the set corresponding to $x > 0 \to y_1 > 0$, by semantics of implication. Another contract, $\widehat{C_1^{x,y_1}} = (True, y_1 = 2)$ is one possible refinement of C_1, that is $\widehat{C_1^{x,y_1}} \preceq C_1^{x,y_1}$. On interface contracts we prefer the word refinement to the word dominance.

Another contract, $C_1^{x,y_2} = (x > 5, x > 0 \to y_2 = 7)$, expresses constraints on the variable y_2. We can put the contracts in conjunction and obtain:

$$\begin{aligned}
C_1^{x,y_1,y_2} &= C_1^{x,y_1} \sqcap C_1^{x,y_2} &&= \\
&= (x > 0 \lor x > 5, (x > 0 \to y_1 > 0) \land (x > 0 \to y_2 = 7)) &&= \\
&= (x > 0, \quad x > 0 \to (y_1 > 0 \land y_2 = 7)) \\
\widehat{C_1^{x,y_1,y_2}} &= \widehat{C_1^{x,y_1}} \sqcap C_1^{x,y_2} &&= \\
&= (True \lor x > 5, (y_1 = 2) \land (x > 0 \to y_2 = 7)) &&= \\
&= (True, (y_1 = 2) \land (x > 0 \to y_2 = 7))
\end{aligned}$$

Let $C_2 = (y_1 < 10, y_1 < 10 \to z \neq 0)$ be another contract and redefine the two contracts above as $C_1 := C_1^{x,y_1,y_2}$, $\widehat{C_1} := \widehat{C_1^{x,y_1,y_2}}$. We can put the contracts in parallel composition:

$$\begin{aligned}
C_{\|} &= C_1 \parallel C_2 &&= \\
&= (x > 0, x > 0 \to (y_1 > 0 \land y_2 = 7)) \parallel (y_1 < 10, y_1 < 10 \to z \neq 0) &&= \\
&= ((x > 0 \land y_1 < 10) \lor \neg G_{\|}, \quad G_{\|}) \\
&\qquad \textit{...where } G_{\|} = (x > 0 \to (y_1 > 0 \land y_2 = 7)) \land (y_1 < 10 \to z \neq 0) \\
\widehat{C_{\|}} &= \widehat{C_1} \parallel C_2 &&= \\
&= (True, (y_1 = 2) \land (x > 0 \to y_2 = 7)) \parallel (y_1 < 10, y_1 < 10 \to z \neq 0) &&= \\
&= ((True \land y_1 < 10) \lor \neg\widehat{G_{\|}}, \quad \widehat{G_{\|}}) \\
&\qquad \textit{...where } \widehat{G_{\|}} = ((y_1 = 2) \land (x > 0 \to y_2 = 7)) \land (y_1 < 10 \to z \neq 0)
\end{aligned}$$

In the example we relied on the reader's imagination to carry out the operators logically and understand how the symbolic abstractions map onto Sets. There are details that need to be made explicit. The first involves unspecified ports when doing contract conjunction. The interface contract C_1^{x,y_1} does neither involve y_2 nor z, so how does it treat conjunction with another contract like C_1^{x,y_2}? We need a variable inverse elimination operator for this, or variable introduction, defined based on the semantics of dominance.

Definition 6. Let $C = (\phi_A(p), \phi_G(p))$ be an interface contract and $p \in \mathcal{V}$ a port. We define *port elimination* of p, written $[C]_p$, as:

$$[C]_p = (\forall p \in \mathcal{D}.\ \phi_A(p), \exists p \in \mathcal{D}.\ \phi_G(p))$$

Elimination is a projection operator: it is existential on guarantees by set inclusion in the straight direction — because dominance only needs one instance for each variable to describe the other dimensions — whereas it is universal on assumptions because it is performed by converse inclusion and it triggers thereby duality.

Elimination of p can be seen alternatively as the supremum of dominance with respect to all contracts deriving from C making the domain $\mathcal{D}$ ranging on p. Variable introduction ought to work consistently, assigning variables to the empty valuation on assignments and to the whole $\mathcal{D}$ on guarantees. This is equivalent to assigning the algebraic $\mathbf{1}$ to introduced variables. In the following we will omit explicit variable introductions, that we will assume without mention whenever needed.

An interface component works like a filter when embedded in an environment. Specifically, assumptions specify when the filter is applicable, guarantees specify with the resulting side effects what the range of its application is. In system design, interface contracts are used to specify either viewpoints or component behaviours. In the first case they are used to specify individual aspects of components, they are combined using conjunction. Different aspects might concern functionality, timings, security, etc. In the second case interface contracts are used to specify the interface behaviour of entire components, they can be the result of the conjunction of different viewpoints and are expected to be combined using parallel composition, thus accounting for mutual cooperation.

The different usage of contracts for system design is shown in example 5 where, implicitly, a procedure of definition for contracts of different viewpoints is proposed, with refinement possibility, conjunction and parallel composition with other contracts. The following example shows the last passage of parallel composition on a simpler case, with the possibility of making the result explicit and getting a stronger intuition for this important operation.

Example 6

Let $C_1 = (x > 0, x > 0 \to y > 0)$ and $C_2 = (y > 0, y > 0 \to z > 0)$ be two interface contracts over $\mathcal{V} = \{x, y, z\}$ and $\mathcal{D} = \mathbb{R}$.

Their parallel composition is:

$$\begin{aligned} C_1 \parallel C_2 &= (x > 0, x > 0 \to y > 0) \parallel (y > 0, y > 0 \to z > 0) \quad = \\ &= ((x > 0 \wedge y > 0) \vee \neg((x > 0 \to y > 0) \wedge (y > 0 \to z > 0)), \\ &\quad ((x > 0 \to y > 0) \wedge (y > 0 \to z > 0)) \quad = \\ &= ((x > 0) \vee (y > 0 \wedge \neg z > 0)), \\ &\quad (x > 0 \to y > 0) \wedge (y > 0 \to z > 0))) \end{aligned}$$

The idea of parallel composition is on combining interface contracts as if their interface components would be to combine, connecting the outputs of one component to the inputs of the other. The presupposition for this to be a sensible interpretation is that inputs (and outputs) are those ports that are kept unconstrained (respectively, constrained) by the contract's guarantees. Matching input/output is done through equality over port names; we keep details off exposition for simplicity.

In the previous example, given a contract taking x, returning y and one taking y, returning z we obtained a contract having x as input and z as output. It is not clear by the final equation what it does though. To see this we shall need to get rid of the port y, which can be seen as an internal port and thus removed with its relative constraints. We do it considering the contracts with less dominance of all, which goes to variable elimination as per definition 6.

Example 6 (*cont.*)

...

The contract that we obtain from eliminating y from $C_1 \parallel C_2$ is:

$$\begin{aligned}
C_y^{\parallel} = [C_1 \parallel C_2]_y &= (\forall y \in \mathbb{R}.\ (\ (x > 0 \vee y > 0) \wedge (x > 0 \vee \neg z > 0))\ ),\\
&\qquad \exists y \in \mathbb{R}.\ (\ (x > 0 \rightarrow y > 0) \wedge (y > 0 \rightarrow z > 0))\ ))\\
&= ((x > 0 \vee \forall y \in \mathbb{R}.\ (\ y > 0\ )) \wedge (x > 0 \vee \neg z > 0)),\\
&\qquad ((x > 0 \rightarrow \mathit{True}) \wedge (\mathit{True} \rightarrow z > 0)) \vee \qquad [\text{by Shannon Expansion True}]\\
&\qquad \vee ((x > 0 \rightarrow \mathit{False}) \wedge (\mathit{False} \rightarrow z > 0))) \qquad [\text{by Shannon Expansion False}]\\
&= ((x > 0 \vee \mathit{False}) \wedge (x > 0 \vee \neg z > 0)),\\
&\qquad ((\mathit{True}) \wedge (z > 0)) \vee ((\neg x > 0) \wedge (\mathit{True})))\\
&= ((x > 0) \wedge (x > 0 \vee \neg z > 0),\quad (z > 0 \vee (\neg x > 0))\\
&= (x > 0,\quad x > 0 \rightarrow z > 0)
\end{aligned}$$

Using variable elimination we obtain a contract that is the best among the less restrictives (by definition of supremum) and thereby such that however the interface components would be of C_1 and C_2 internally to $C_y^{\parallel} = (x > 0,\ \ x > 0 \rightarrow z > 0)$, the guarantees will not fail to hold. This is a very strong point and also provides the right means to read the final contract as a whole.

Now that the haze is fading around the framework, we push it on exploring different paths of composition. Let us see for instance, how parallel composition can be performed on contracts mutually constraining one another's inputs:

Example 7

Let $C_1 = (x > 0, x > 0 \rightarrow y > 0)$ and $C_2 = (y > 0, y > 0 \rightarrow x > 0)$ be two interface contracts over $\mathcal{V} = \{x, y\}$ and $\mathcal{D} = \mathbb{R}$.

Their parallel composition is:

$$
\begin{aligned}
C_1 \parallel C_2 &= (x > 0, x > 0 \to y > 0) \parallel (y > 0, y > 0 \to x > 0) &&= \\
&= ((x > 0 \land y > 0) \lor \neg G, \quad G) &&= \\
&\quad \textit{...where } G = (x > 0 \to y > 0) \land (y > 0 \to x > 0) = (y > 0 \leftrightarrow x > 0) \\
&= ((x > 0 \land y > 0) \lor \neg(y > 0 \leftrightarrow x > 0), \quad (y > 0 \leftrightarrow x > 0)) &&= \\
&= ((x > 0 \lor y > 0), \quad (y > 0 \leftrightarrow x > 0))
\end{aligned}
$$

Seeing the relative interface components as a closed system we can get rid of x and y by a double elimination on the lines proposed in the previous example and definition 6, obtaining $C^{\parallel}_{x,y} = (False, True)$, which embeds in every environment and guarantees everything about variables. This is the algebraic **1** by no accident and it is a consequence of us deciding elimination based on dominance's suprema: for an arbitrary implementation choice of the individual interface contracts it cannot be ensured that incompatibility will not be there, hence variable elimination cannot provide anything different than the maximum of the lattice induced by dominance.

Combining contracts using mutual relations is common when the goal is to model feedback components[82][76]. We will see that this type of construction will turn out to be useful when the verification or dependability analysis of complex Cyber-Physical Systems with feedback is probed (cf. section 6.1.2).

We conclude the section by showing what happens when contracts are not compatible, namely when their output values and input values do not match on same ports. We use a revisiting of example 6:

Example 8

Let $C_1 = (x > 0, x > 0 \to y > 0)$ *and* $C_2 = (y \leq 0, y \leq 0 \to z > 0)$ *be two interface contracts over* $\mathcal{V} = \{x, y, z\}$ *and* $\mathcal{D} = \mathbb{R}$.

Their parallel composition is:

$$
\begin{aligned}
C_1 \parallel C_2 &= (x > 0, x > 0 \to y > 0) \parallel (y \leq 0, y \leq 0 \to z > 0) &&= \\
&= ((x > 0 \land y \leq 0) \lor \neg((x > 0 \to y > 0) \land (y \leq 0 \to z > 0)), \\
&\quad ((x > 0 \to y > 0) \land (y \leq 0 \to z > 0))
\end{aligned}
$$

Since $x \leq 0$ *iff* $\neg(x > 0)$ *we can use Shannon expansion to push down the universal and existential quantifier on assumptions and guarantees, obtaining:*

$$
\begin{aligned}
[C_1 \parallel C_2]_y = (&(x > 0 \land False) \lor \neg((x > 0 \to True) \land (False \to z > 0)) \land \\
&\land ((x > 0 \land True) \lor \neg((x > 0 \to False) \land (True \to z > 0)), \\
&((x > 0 \to True) \land (False \to z > 0)) \lor \\
&\lor ((x > 0 \to False) \land (True \to z > 0)) &&= \\
&(((False) \lor \neg((True) \land (True))) \land \\
&\land (((x > 0) \lor \neg((\neg x > 0) \land (z > 0))), \\
&((True) \land (True)) \lor \\
&\lor ((\neg x > 0) \land (z > 0))) &&=
\end{aligned}
$$

$$((False) \ \wedge ((x > 0) \vee \neg(z > 0)), \ \ True) \qquad = \\ (False, \ \ True)$$

This shows that from two incompatible interface contracts we obtained a new interface contract that ensures everything possible, but embeds in no environment. For example any interface component satisfying the constraint $x > 0 \wedge z = 5$ would correctly implement the contract, but it could not be embedded in any context. This is different from what we saw for example 7, because here there is no feedback that variables can agree upon in order to make guarantees consistently false. Here the $False$ assumption means that no interface component can be constructed from the combination of other two satisfying the interface contracts, thus the composition is defective. We say in this case that the two interface contracts are *incompatible*.

The last mention is on saturation. For a pair (A, G) to be a contract, we saw that the complement of the assumption has to be part of the contract's guarantees. This was translated symbolically by pushing the assumption to the guarantees by means of an implication. Although other expositions give the condition as an option to push on the interface contracts whenever necessary to guarantee closure properties or such, we placed it in the very definition of a contract (definition 1). We now propose to streamline notation and instead of writing $C = (A, A \rightarrow G)$ we write $C = (A, G)$. For instance, instead of writing $C = (x > 0, x > 0 \rightarrow y > 0)$ we write $C = (x > 0, y > 0)$, but importantly this would still mean A contract $C = (x > 0, x > 0 \rightarrow y > 0)$. This will not change our formal definition:

** it is only an abuse of notation to avoid cumber **

We will use this streamlined notation on in section 5.3.3.

3.3 Contracts, issues and interface theories

Contracts provide a very general framework based on what is often called the assume/guarantee style of reasoning. An alternative approach is provided by interface theories [45], which instead of treating the two aspects of the system using separate sets of behaviours, a distinction is only made on the separation between input and output variables — which we recall is not explicit in the contract framework as we presented it — and a single first-order relational formula used for the specification of behaviours at the interface level.

Contract modeling is a newer concept than interface theories and offers an alternative formalism that is still very similar to straight interface reasoning. Only recently their difference was sharply drawn, in [76]. One noteworthy component of the paper is a transformation from interfaces to contracts, by a function — or a functor someone would say — that considers the relational formula of the interface to play the role of the contract guarantee and its projection over the input variables to play the role of the assumption. It is not in the scope of this section to discuss formal details already present in the paper; we would like instead to put the stress of our mention to this transformational correspondence between the two formalisms, which is found to preserve most of the interesting properties, between compatible components especially.

Although beneficial in many directions, the contract formalism has the important drawback of not being able to detect incompatibilities before composition is performed: the only way to see whether two contracts cannot find an embedding is by composing them and check they have the empty assumption. This is an acknowledged problem and research paths have been taken lately to check contract compatibility more efficiently [16].

Another issue concerns contract saturation. One of the nicest things about interface contracts is that they provide a formal basis independent from the underlying language $\mathcal{L}$. Almost. One property that we omitted is that $\mathcal{L}$ should be closed under complement. By definition 1 every contract must include the complement of the assumptions in its own guarantees; of course this would not be possible with languages that cannot express complement. The issue might look like a minor concern at first, but unfortunately many applications of the contract framework lay in the cyber-physical domain and thus necessitate somehow continuous or differential semantics [82]. It is thus common to use languages, such as that of hybrid automata, for which complement is not generally feasible. One possible way out is by omitting the saturation property, but then some properties of the framework that rely on it no longer hold. This is an important aspect to consider, based on the application at hand. In the know of the problem we will not concern about it any more: we will get over it and stick to the saturated contracts of definition 1.

3.4 A novel approach to system design

Contract based design has a chief fundamental advantage to other techniques, which is also its goal, that is modular system development. Starting from a high level system specification, engineers can differentiate on different cooperative components to be part of the whole system and manage the development of each component independently from the rest of the system, as long as they ensure the satisfaction of the component contract by their implementations. Checking implementability in a modular fashion is typically performed, formally, by checking contracts' composition dominance to the system contract and then contract satisfaction of the single interface components: given a decomposition of the top-level contract C in contracts C_1, C_2, C_3 and an implementation of each, say M_1, M_2, M_3, the following formulas are applied (not necessarily all at once):

$$C_1 \parallel C_2 \parallel C_3 \preceq C$$

$$M_1 \models C_1 \quad \wedge \quad M_2 \models C_2 \quad \wedge \quad M_3 \models C_3$$

The decomposition might further be applied to contracts C_1, C_2 and C_3 separately, down to interface contracts of low tier of abstraction.

Moreover, the very implementation of a contract can be split among different teams in charge of different aspects of the system, by developing different viewpoints independently, merged to refine the needed contract. In formal terms, given a viewpoint separation of, say C_1, in e.g. C_1^{safety}, C_1^{timing}, C_1^{energy} and $C_1^{reliability}$

with corresponding implementations M_1^{safety}, M_1^{timing}, M_1^{energy} and $M_1^{reliability}$, the following are checked in replacement for $M_1 \models C_1$:

$$C_1^{safety} \sqcap C_1^{timing} \sqcap C_1^{energy} \sqcap C_1^{reliability} \preceq C_1$$

$$M_1^{viewpoint} \models C_1^{viewpoint}$$

...where $viewpoint \in \{safety, timing, energy, reliability\}$

The nice part of the process is that, again, integration is granted to be correct by construction, if the single contracts at the low tier are satisfied by the given interface components (see also [12] for more advanced methods for obtaining smaller contracts wrt dominance).

Continuing our methodological discussion, after the anticipations of chapter 1 it should no surprise that this process is overall adequate in the untouchable ideal world of mathematics only. When real world component implementations are deployed on a physical ecosystem, contingencies can bring systems to failure and break components' guarantees regardless of assumptions being satisfied. Errors and faults can propagate to other components too and take the system to its crash. In chapter 1 we argued that fault injection is a powerful weapon to detect system errors, and there we also saw that by using model-based techniques it is possible to analyze mathematical models through extensions and possibly construct fault trees, or other engineering artifacts, in an automatic way. Integration of fault injection with the contract framework has hardly ever been tackled by the literature — we discussed [20] as a valid ongoing proposal; part of the contribution of this thesis is to show how contracts can be used very naturally to achieve scalability within analysis processes of complex Cyber-Physical Systems, including dependability assessments.

The theory of contracts is not only a mathematical tool, but is a mathematical basis that suggests a new way of thinking design. Besides modularity that we already talked about, there is also a fundamental concept of hierarchy that undergoes the framework and provides a vigorous and prominent confront to issues of scale. Integration of components in any compatible environment provided the assumptions plus separation of concerns deriving from the development of single isolated components and viewpoints are cherished characteristics of remarkable value for industry. Also, design by contracts accommodates verification well, because hierarchical design allows for compositional reasoning and relative techniques to tame complexity and state-space explosion: properties about the systems as a whole can be deduced from properties of its isolated parts. This will be our attitude towards contracts: we will acknowledge their capacities not only within system design in a classical sense, but even in combination with SOA-based collaborations.

The interface aspects beneath service orientation can be cast to the contract formalism and the different parts of the architecture can be too, including submissive agents such as the network where the agents exchange their interactions. As we mentioned back in section 2.2.2 in fact, interface contracts can be utilized for the reappraisal of formalized services [26]. Recall that in the cited paper the behaviour of components is defined by input/output relations over streams of messages, compositionally blended together using concepts of composition and behavioral refinement, which are by all means resemblant to ours of parallel composition and dominance. Furthermore the work in [26] defines services as partial behaviours, whereas components are required to expose an output for each possible input. This is in line with the different notions of contracts and implementations that are typical of contracts.

In a strong sense the work in [26] can be seen as an alternative interface theory, where input and output variables are streaming channels, which are related by a relation of interface behaviour. It should be clear in the informal sense from our previous discussion that the difference between formalisms of this kind and contracts is only one property preserving function far from being none, and it thus shouldn't be deemed dissimilar in substance. From this direct matching of contracts from the service formalism of [26] we build confidence that studying services at the interface level is a perfectly legitimate and sensible choice. Moreover, using contracts as a formalism consents to capture the service behaviour of interacting systems coherently with the latest responses to embedded systems' design challenges, an important aspect for assessing the value and appropriateness of the techniques developed in this thesis.

Chapter 4

The tool XSAP

XSAP is the eXtended Safety Analysis Platform, designed for the assistance of safety and design engineers in the development of digital embedded systems. It is extension and reappraisal of FSAP/NuSMV-SA [23], developed by the same authors in fact (cf. section 1.1).

In this chapter we will present the tool from different standpoints and levels of detail. As a gentle introduction, we will provide an overview of the functional facet of the system, including features and rationale. We will then give an overview of the formal techniques that lie beneath the tool implementation. Even though our usage of the tool will be confined to its user interface, understanding the technology is critical, besides scientific motivations, in order to evaluate their applicability.

This will bring us to discuss their dissemination within industry and their additional spreading potential with XSAP. Finally we will present the practicalities of the tool at the interface level, in order to get confidence in preparation of chapter 6, where we will render its application on the use cases of chapter 5.

4.1 Functionality

From the functional standpoint, XSAP is a model-based fault injection tool, in the sense of section 1.2. In the same reference section we also discussed that any creditable platform for the software or hardware fault injection should come with a *fault injector*, a *library of admissible faults*, a *workload generator* of admissible system inputs and *a monitor* to collect and present the results of the analysis. We also hinted that this could extend to all fault injection methodologies, thereby including model extension.

From a functional point of view, XSAP is to be provided with a model of the nominal system (i.e. a description of how it should behave, supposedly) and a model of injections that specify the possible faults that may jeopardize system safety. The two models are fed to the tool as two separate files. This distinction between system model and injections is a clear strength of XSAP, essential for model-based

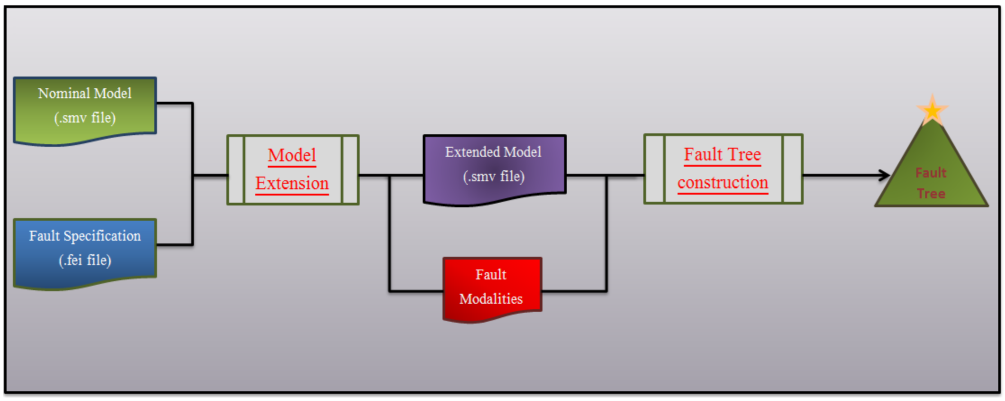

Nominal Model (.smv file)
Fault Specification (.fei file)
Model Extension
Extended Model (.smv file)
Fault Modalities
Fault Tree construction
Fault Tree

Figure 4.1: Fault Tree Generation: XSAP takes the nominal model, the fault description and constructs the Fault Tree checking the Top-Level Event against the extended model.

approaches to sustain the integration with the design workflow[1].

Behind closed doors, XSAP makes use of patterns to let specify the fault modalities in the injection file. This is done by an optionally extensible set of fault patterns expressed as state machines in XML format. As an example, a stuck-at-0 fault is specified in the library as a state machine that transits between two states, nominal and faulty, by setting the target variable to 0 and keeping that value fixed throughout execution thereafter. About this and how to customize this process we will spend a bit more of words later.
After that, automatically, XSAP can be instructed to extend the nominal model and, triggering fault happenings, to construct fault trees and FMEA tables of minimal size. Figure 4.1 shows schematically the procedure to construct a fault tree.

Recalling the common characteristics of fault injection processes that we mentioned above, in the model-based injection performed by XSAP we can identify the *fault injector* as XSAP itself, combining the nominal model and the fault specifications (this is achieved using the python script `extend_model.py`, provided with the tool). The *library of admissible faults* is the extensible source of fault patters *behind the closed doors*. The *workload generator* is embedded in the semantics of the model, i.e. the transition system on a Kripke structure and the monitor can be seen as the model-checking procedure itself, that results in property violations under faults, later translated to fault trees and FMEA tables (this is achieved using the python script `compute_ft.py`/`compute_fmea_table.py`). More formal details later.

XSAP is a complex tool and is supported by the long-aged algorithms of NuSMV. All of its history is about embedded systems design and verification and its de-

[1]The benefits, indeed, are manifold. One is certainly being able to independently grow the nominal system without caring about faults, another is having, possibly, several models of injection: the simplest to be applied at the earliest stages of design and the others, refined, later.

velopment has been supported and motivated by industrial concerns, such as the automatic construction of engineering artifacts. This gave us strong assurances regarding its adoption, that met our prospects well.

In addition to the construction of Fault Trees and FMEA tables, XSAP also supports failure propagation using the concept of Timed Failure Propagation Graphs (TFPGs) [2]. The application of this part of XSAP to our quest is beyond the scope of this thesis and will not be developed further. We refer to [54] for an usage account of XSAP under those terms.

4.2 Formal Nuts and Bolts

To understand XSAP we need to understand model checking first. We assume the reader has some basic knowledge about the subject and only recall key concepts of the mathematical machinery. Then we will have a short digression on the specific functionalities featured by NuSMV/nuXmv, as they will have an impact on our experimental treatment of chapter 6. Finally we will provide a gist of the algorithms underlying the Fault Tree construction, as per acquirement from the literature.

4.2.1 Concepts of Model Checking

For this light presentation of concepts we will follow the common practice of defining the mathematical structure at use, its formal semantics and a logic upon that structure. We will briefly highlight what is possible to achieve with such a logic and why is important for the the verification of system design.

We will highlight what in this framework suits our needs and what doesn't, stressing similarities and differences that we find between traditional reactive systems and large-scale Cyber-Physical Systems, that are the target of our study.

Kripke semantics and Language

The mathematical structure of interest is the so-called *Kripke structure*. A Kripke structure is a quadruplet (S, I, R, L) defined over a set of boolean variables $\mathcal{V}$ where:

- S is a finite set of objects, called states
- $I \subseteq S$ is a set of initial states
- $R \subseteq S \times S$ is a non-deterministic transition relation between states
- $L : S \to \mathcal{P}(\mathcal{V})$ is an injective map, called labeling, that says, for each state, which among the available boolean variables are true.

The Kripke structure can be easily generalized to atomic propositions of any kind, by taking their boolean abstractions as boolean variables (e.g. use v_1 as a boolean variable, and put $v_1 \leftrightarrow x = 3$).

The intuitive interpretation of a Kripke structure is that, starting from the states in I, the reactive system evolves indefinitively, according to R. For the latter reason it is typically required not to have dead-ends, i.e. that every state have at least one successor. So formally, the semantics of a Kripke structure is given by sets of *runs*, where runs are infinite sequences of states $\{s_i\}_{i\in\mathbb{N}}$ such that $s_0 \in I$ and $\forall i \in \mathbb{N}.\ (s_i, s_{i+1}) \in R$. Finite and infinite initial segments of runs are called *traces.*

Kripke structures are very much suitable to be checked against modal logics. Here we focus on the popular Linear-time Temporal Logic (LTL) [79], that is an extension of classical propositional logic — which already had $\neg$, $\wedge$, $\vee$, $\rightarrow$ — with temporal operators on traces (**X** and **U**). Temporality allows to travel along the Kripke structure's transition relation and is semantically expressible by tracking the system evolution on traces. Formulas of LTL are thus defined on traces, whereas sub-formulas on future sub-traces.

The temporal extension of propositional logic is given by the following two temporal operators:

- The neXt unary operator ($\mathbf{X}\phi$), that expresses the validity of a formula ϕ in the successor state of a trace. Formally, $\mathbf{X}\phi$ is satisfied in state s_i of path π iff ϕ holds in state s_{i+1} of the path π.
- The Until binary operator ($\phi\mathbf{U}\psi$): in a future sub-trace where a formula ψ is somewhere true, it expresses the validity of a formula ϕ in the initial sub-trace preceding the occurrence of ψ. Formally, $\phi\mathbf{U}\psi$ is satisfied in state s_i of path π iff there exists $j \geq i$ such that (1) ψ holds in state s_j of path π and (2) for all k within, ϕ holds on the path (i.e. ϕ holds in all s_k of π such that $i \leq k < j$).

In terms of the neXt and Until operators, the temporal operators of Finally (**F**), Globally (**G**) and Release (**R**) are definable, with the respective intended meaning of (**F**) the eventual occurrence of an event, (**G**) the universal occurrence of an event and **R** the unconstraining of the system upon an event (e.g. drink **R** thirst) [2].

LTL allows to formalize temporal properties of systems in different domains, as the following propositions show[3].

- *The variable x is never 0*:
$$\mathbf{G}\neg(x = 0)$$
- *Whenever the client requests a service, the service will be provided within one and three steps*:
$$\mathbf{G}(\mathit{request} \rightarrow (\mathbf{X}\mathit{served} \vee \mathbf{XX}\mathit{served} \vee \mathbf{XXX}\mathit{served}))$$
- *The aircraft's doors are kept closed while the engines are on, but they will not be kept closed forever*:
$$(\mathit{doorsClosed}\mathbf{U}\neg\mathit{enginesOn}) \wedge (\mathbf{F}\neg\mathit{doorsClosed})$$

[2]The definitional equivalence: $\mathbf{F}\phi := \texttt{TRUE}\mathbf{U}\phi \quad \mathbf{G}\phi := \neg\mathbf{F}\neg\phi \quad \phi\mathbf{R}\psi := \neg(\neg\phi\mathbf{U}\neg\psi)$.

[3]Obviously, a sensible modeling of atomic propositions in a Kripke structure prior to the definition of LTL formulas, that we omit relying on the reader's intuition, would be mandatory.

The model-checking problem of LTL asks whether a given LTL formula ϕ holds in all traces of a given Kripke structure $\mathcal{K}$. If this is the case, it is common to write $K \models \phi$. [4].

Interest in Model Checkers and LTL

An LTL model checker is a tool implementing a decision procedure that, given the specification of a Kripke structure and an LTL formula, it returns YES or NO based on the satisfaction of the formula in the Kripke structure. If the model-checker finds a disatisfaction of the formula, it returns a counterexample, namely a trace representing an evolution of the Kripke structure leading to falsification.

LTL has wide application in the verification of systems subject to temporal evolution and it has been employed in the past for the verification of software, concurrency protocols and reactive systems. Its popularity is due to the simplicity, compactness and expressiveness of its temporal formulae; differences in model checkers are determined by the language to construct the Kripke structure — which normally depends on the application domain — and implementation performance. Needless to say, there exists even other logics and variants that apply to Kripke structures or rather, just for a reference, to other interpretations of Kripke frames. For as much as we will be concerned to model checking, knowing about trace semantics, satisfaction and counterexamples generation will be enough.

As an anticipation, we point out that the full power of LTL will not be necessary to our direct aims, but it will be at support. In chapter 6 we will assess the dependability of systems using the concept of an invariant, which is easier to roll off and apply to the construction of engineering artifacts. Besides, we value the presentation of LTL as a means for the understanding of the logical concept of temporality.

Invariants represent safety and and are optionally representable in LTL in the universal form $\mathbf{G}\phi$, having ϕ propositional. They represent something that must hold, logically, in every state of any trace and are characterized by locality, i.e. they do not involve the future of the trace, only single states universally in all traces. This is what makes them amenable to automatic construction of artifacts, because checking invariant specifications reduces to solving a reachability problem over the Kripke structure. This can be handled by sophisticated and specialized techniques founded on general induction in a constructive sense.

The usage of temporal formulae in our given scenarios will be part of the contract formalism. This is a need, as we saw in other chapters, for a choice of sensible modeling of cyber-physical architectures with services and represents our first difference with respect to standard model-checking/dependability activities on systems.

[4]Checking this algorithmically means to provide temporal satisfaction guarantees on possibly infinite evolution traces. Fortunately, the Until operator comes with a so-called *fixpoint semantics* and is recursively definable as $\phi\mathbf{U}\psi := \psi \vee (\phi \wedge X(\phi\mathbf{U}\psi))$. Working in the class of propositional boolean formulas ordered by logical consequence, thus a lattice, we have strong guarantees of well-foundedness of its semantics.

Pattern-based languages

The second divergence with standard activities is that our final target is industry. Practitioners are known to lose confidence and reliability as language's expressive power grows, therefore it is preferable to let them have only few patterns available for their specifications, and in human readable form. This solution includes more controllable specifications and also, optionally, dedicated treatments for the specification of system requirements. One of those might be, for example, that invariant property specifications could be supplemented with the possibility of deriving fault trees relative to their negations.

The Block-based Contract Language (BCL)[52] can be seen as an alternative to the language of LTL that accounts for pattern-based specification of contracts. In the specific case of the presentation paper, the authors use patterns to specify assumptions and guarantees on Matlab/SimuLink blocks and verify their compatibility and compositionality using simulation-based techniques. We are not interested in the whole framework of BCL but on its pattern-based language, an oversimplified fragment in fact, to get the gist of the usage of patterns.

In BCL events happen in time instants, in time intervals or against timeouts. Conditions, besides events, include boolean expressions of comparison and boolean compositions of those under conjunction, disjunction, implication and negation.

Now, if **E** is an event and **C** a condition, one can specify BCL patters as

- [**E**] **happens within** [3s], expressing time-bounded firing of an event
- [**C**] **holds**, expressing a condition holding
- **Everytime** [**E**] **then** [**C**], expressing an event dependent condition
- [**C**] **always**, expressing persistence (invariance)

There are more, with variants and generalizations, but we limit ourselves to those for simplicity.

We would like to point out that there certainly exist other pattern-based languages. We picked BCL among all because we envisioned a possible future work of integration with the Matlab/SimuLink framework it provides. However our contributions are independent to the specific logic language at use and we believe that experiencing patterns would at least help making our objectives with respect to industry clearer. In the specification of our use case example of section 5.3 we will use BCL as our property specification language exactly for the reasons we outlined in this paragraph and we will propose an informal translation of our BCL specifications to LTL formulae (comprising invariants) before feeding them to the model checker engine for artifact construction.

4.2.2 The NuSMV/nuXmv base

We already saw in the previous sections that the tool XSAP is a safety-analysis tool based and integrated with the nuXmv model checker [30]. Following this view, we could rethink the functionality of XSAP as split into two parts: the model checker and the safety analyzer. After the construction of the Kripke structure common to both, the model checker allows to verify temporal specifications, the safety analyzer to construct fault artifacts from invariant specifications. We defer the modeling of the Kripke structure to section 4.3.1 and the artifact construction to section 4.2.3. In this section, instead, we focus on the model-checking possibilities of nuXmv. Again, we will not be exhaustive but assume that a basic knowledge is already part of the reader's background.

Traditionally, model checkers arrange the internal representation of the state space of a Kripke structure in either explicit or symbolic form. The model checker nuXmv goes for the symbolic option. Instead of enumerating all the states in the model one-by-one, nuXmv represents many, collectively, as boolean formulas. A formula such as $b_1 \rightarrow x = 3$, for example, represents all states where variable b_1 is false and variable x is arbitrary, plus that single state which interprets variable b_1 as true and variable x identical to 3. The symbolic representation in adoption of nuXmv has a number of advantages, such as performance boost and more natural encoding of system variables in the system.

A complete treatment of the subject is way beyond the scope of this thesis, but at least we need to mention the four chief four strategies used by nuXmv for carrying out the symbolic model checking. These are BDD-based checking[29], SAT-based BMC[17], SMT-based BMC[5][8] and IC3[34][35]. The brief discussion in this section will make an effort to highlight strengths and weaknesses of each technology or else, to be more precise and perhaps fair, the different circumstances where each fits best.

BDD-based model-checking

Binary Decision Diagrams (BDDs) [28] are highly tuned data structures for the efficient manipulation of boolean formulas, both in terms of memory and performance. Since their acknowledged establishment, they have been used in model checking to encode Kripke structures, comprising initial states, transitions and their labeling, as boolean formulae over the structures' state space. Again, if the involved variables or propositions are not boolean, some form of abstraction or transformation can make them so.

Concretely, initial states of the structure are represented as the boolean conjunction of the initial constraints and the transition relation as the boolean relation between symbolic states and their future counterparts, traditionally denoted by primed variables. The labeling function is embedded in the symbolic representation. Put as simple as it is, collection of states evolve semantically from the initial representation by means of the transition relation realized in boolean logic using perfect boolean abstraction.

Take as an example the Kripke structure generated by a single boolean variable f, initially $False$, that switches from $False$ to $True$ in a non-deterministic fashion and remains such from then onward[5].

The corresponding formal system would be defined as

$$\mathcal{K}_f = \langle S = \{1,2\}, I = \{1\}, R = \{(1,1),(1,2),(2,2)\}, L = \{(1,\emptyset),(2,\{f\})\}\rangle$$

Symbolically, one would represent the entire Kripke structure $\mathcal{K}_f$ as

$$\begin{aligned}
I_{BDD}(\mathcal{K}_f) &= \neg f \\
R_{BDD}(\mathcal{K}_f) &= (\neg f \wedge \neg f') \vee (\neg f \wedge \neg f') \vee (f \wedge f') && = \\
&= \neg f \vee (f \wedge f') && = \\
&= f \rightarrow (f \wedge f') && = \\
&= f \rightarrow f'
\end{aligned}$$

If we would like to know all the models of the Kripke structure over the variables involved in it — only f in our simple example — we can inductively reiterate application of the symbolic abstraction of the transition relation $R_{BDD}(\mathcal{K}_f)$ starting from the symbolic representation of the initial states $I_{BDD}(\mathcal{K}_f)$, up to a point of idempotence, which has strong mathematical guarantees to be reached:

$$\begin{aligned}
F_0 &:= I_{BDD}(\mathcal{K}_f) = \neg f \\
F_1 &:= F_0 \vee (\exists f \in F_0.\ R_{BDD}(\mathcal{K}_f)) \\
&\quad = \neg f \vee \exists f_{pre}.\ (\ \neg f_{pre} \wedge (f_{pre} \rightarrow f)\ ) \\
&\quad = \neg f \vee True = True \\
F_2 &:= F_1 \vee (\exists f \in F_1.\ R_{BDD}(\mathcal{K}_f)) = True
\end{aligned}$$

The fixed point symbolic representation F_2 indicates that both values of are possible for f during system evolution. The [oversimplified] procedure that we described shows how the reachability problem is tackled, symbolically, using boolean function representational utilities such as BDDs.

Model checking is way off a harder beast, both computationally, theoretically and in terms of implementation. We are not particularly interested in model-checking techniques in this thesis. What is important is having a rough idea about the symbolic representation of Kripke structures, and what BDDs are great at, namely manipulation of booleans. We will take on the reachability problem in section 4.2.3, to explain, conceptually, how dependability artifacts are constructed in XSAP.

SAT/SMT-based Bounded Model-Checking

Although BDD-based techniques are efficient, they demand a complete construction of the state space as an efficient, yet possibly enormous, BDD. To avoid the complete construction of such BDD, Bounded Model Checking techniques (BMC) are employed that only partially and incrementally unroll the transition relation.

[5]Such a Kripke structure might be interpreted as a state machine describing a permanent fault, that once active it never gets fixed.

BMC techniques are distinguished in SAT-based and SMT-based. In short, SATisfiability is the problem of checking whether a boolean formula can be made true by a suitable valuation of variables, Satisfiablity Modulo Theory is the problem of checking whether boolean combination of constraints in some decidable theories admit a variable assignment so to make a quantifier-free theory formula true. As a matter of implementation, it has been discovered that SMT solvers can be seen as improvements to SAT solvers with domain-specific reasoning skills, playing their variable assignment's satisfiability game of on theories instead than on atomic propositional valuations. We refer to [9] for a comprehensive discussion of the subject.

In both SAT and SMT cases the Kripke structure is built incrementally starting from the initial states and on by unrolling of the transition relation, gradually incrementing the bound of the procedure up to a certain maximum bound k.

To do this, temporal properties are put in conjunction and solved as an incremental SAT or SMT problem. The technique is bounded and as such is not complete: this is the price for avoiding the complete construction of the whole model. It is typically used to find counterexamples of increasing length fast and it owes much of its popularity to the constant parallel development of latest advances in SAT and SMT solvers.

Model-checking with IC3

The technique underlying IC3 tries to find system's inductive invariants, and it does it incrementally driven by over-approximations of the Kripke semantics of the system. To do that, at implementation it separates between reasoning in the boolean domain and reasoning at the theory level, exploiting theory-specific techniques for abstraction refinement provided by SMT solvers in order to find the inductive invariants for refinement[35].

Importantly, the incremental aspect is not found on monotonic unrolling of the transition relation, but on the refinement process upon the over-approximation. This leads to a complete methodology comparable to the BDD-based approach, albeit more effective (or even uniquely feasible) in presence of complex theories where theory reasoning is unavoidable on the model.

On the availability of different techniques

The significance of knowing about the different algorithmic availabilities of the state space exploration procedures is twofold. First it will make sense out of the different performance that we will get out of fault tree construction; this is of practical significance. The second interest relates to how we deal with system heterogeneity. Because different actors in the SOA will be given different roles and functionalities, and because the particular designs will be different, it is to be expected that different algorithms will fit different domains differently. For example, in the boolean realm we expect BDDs to be the most effective, whereas SMT-based techniques would be better when logical reasoning on theories is required. Having available an adaptable [and evolving] state-of-the-art tool such as XSAP is of huge value when diversity partakes the ground. Moreover we will be able to treasure this aspect even more, because we will advocate its usage in the context of contracts, which are per-se practical solutions to system's heterogeneity. The difference is that whereas contracts work at the interface level, XSAP will work over actual implementations.

On synchronicity of the tool and timings

NuSMV and its successor nuXmv are model checkers explicitly designed for the symbolic analysis of reactive systems, such as embedded devices. Multiple devices can be specified as separate system modules, or components, whose connections to one another are specified by indicating which variables of the one component should be input of the other. This correspondence between inputs and ports is implemented by syntactic substitution of terms, which makes the tool synchronous, in conclusion. In other terms, the system evolves pushing values on ports, with zero-delay and all together.

When Cyber-Physical Systems rely on Service-Oriented Architectures, or on networked architectures in general, synchronous communications are the farthest abstraction to reality. As we saw in chapter 2, timing is a very important aspect in SOA and can have an impact on both system performance and functionality. Our dependability study has thereby the very serious obligation to account for that aspect. In section 6.1 we will outline our answer to this apparent obstruction, not asking designers to enforce asynchronous semantics on the model, but rather drawing a sharp line between system functionality and network communication, using separation of viewpoints in the spirit of contract-based design (cf. chapter 3).

4.2.3 Fault Tree construction in XSAP

In this section we will delineate the Fault Tree construction procedure used by XSAP for the construction of Fault Trees. Our usage of XSAP is off-the-shelf, thus we have no information about its implementation beyond literature investigation. That will be enough, however, to appreciate the technology proposed by the tool. Moreover, since formal techniques for the construction of FMEA tables are alike by all conceivable intents, we will restrict our attention to Fault Tree constructions, without loss of generality. To be more definite, following the lines of [21] we will focus our attention on the BDD-based procedure, confident that the reader can generalize to how this is translated to SAT/SMT-based techniques or IC3.

A first phase injects faults in the model. This is done in XSAP, as more clearly expounded in section 4.3.2, by dedicating specialized *hook variables* to the occurrence of faults and starting nominal or faulty models correspondingly. Then, after model fault extension and provided with a Top-Level Event to reach, the fault tree construction procedure can commence with the aim of selecting all minimal sets of hook variables (the so called *cut sets*) that can lead the Top-Level Event to occur.

Formally, if $\mathcal{F}$ is the set of all the hook variables of the extended Kripke structure $\mathcal{K}_{\hat{\mathcal{F}}}$, minimal cut sets of the Top-Level Event ϕ_{TLE}, whose corresponding reachability invariant would be $\mathbf{G}\ \neg\phi_{TLE}$, are those in

$$\begin{aligned} CS(\phi_{TLE}) = \{\mathbf{FC} \in 2^{\mathcal{F}} \mid \exists s_0, s_1, ..., s_k \in Traces(\mathcal{K}_{\hat{\mathcal{F}}}). \\ s_k \models \phi_{TLE} \wedge \\ \forall f \in \mathcal{F}.\ (\ f \in \mathbf{FC} \leftrightarrow \exists i \in 0..k.\ s_i \models f\ )\} \end{aligned}$$

required minimal by set inclusion. Recall that the expression $s_i \models \phi$ means that in the *i*-th state of the trace, the propositional formula ϕ holds. In other words we look for the minimal sets among all those containing hook variables that can lead, for some trace of the Kripke structure, to the logical satisfaction of the Top-Level Event. Notice that the locality of invariant properties turns out here, in the very definition of cut sets.

The algorithm to find the cut sets relative to a Top-Level Event is conceptually simple, then variants and optimizations are implemented in practice. The main two possibilities is on the search procedure, which can run either forward, from the symbolic representation of the input states, moving the symbolic frontier $\subseteq$-monotonically towards the Top-Level Event according to forward application of the Kripke structure's transition relation, or backwards, from the symbolic representation of the Top-Level Event, still $\subseteq$-monotonically using this time the backward application of the transition relation up to intersection with the initial states. We already approached the forward procedure in section 4.2.2, when we talked about the invariant checking procedure using BDDs.

The forward and backward applications are defined by quantifying away the current and next states from the transition relation. In logical terms, if Q is the set of states at the frontier (regardless whether for the forward or backward procedure), this is symbolically possible by using the formulas

$$\exists x \in Q.\ (\ R(x,y)\ ) \qquad \exists y \in Q.\ (\ R(x,y)\ )$$

Simultaneously, a set of boolean *history variables* characterized by a bijection with the hook variables in $\mathcal{F}$ monitors the evolution of the system, each set to true in case of the corresponding hook variable is set to active throughout the evolution.

After reachability completes we have the symbolic representation of all the end frontiers, heading or tailing paths from the initial states to the Top-Level Event. From those states we extract the history variables (the cut sets) and a minimization procedure is undertaken. If the cut sets extraction is performed by a straightforward variable elimination (i.e. logically quantifying away all variables which are not hook variables), minimization is more involved and needs to be treated separately. We report the BDD-based minimization procedure presented in [80], to provide an intuition to how this can be made in practice[6].

The starting point is a disjunction of BDD representations of the extracted cut sets. This is a fault tree, but it is not minimal by construction. One way to see how the BDD encoding works is through *if-then-else* constructs, binary decisions, not

[6]Just for a reference, minimization in SAT/SMT-based model checking can be performed by adapting the algorithms for minimal unsatisfiable cores extraction [95][37]

accidentally. This follows by a property of propositional logic: given $\phi(x)$ a formula dependent from a variable x, there are two formulae ϕ_A and ϕ_B independent from x such that $\phi(x)$ is logically equivalent to $(x \wedge \phi_A) \vee (\neg x \wedge \phi_B)$ — a consequence of the so called Shannon's decomposition.

In other words the statement says that the formula $\phi(x)$ is equivalent to the expression

$$\texttt{if } x = True \texttt{ then } \phi_A \texttt{ else } \phi_B$$

which is a decomposition able to detach x from the rest of the formula. If one starts producing sub-formulas in this way, recursively removing variables of sub-formulas in a fixed order, a binary tree obtains, whose branches all end up in atomic boolean constants, namely in $True$ or in $False$. The idea is that following branches in the choice of taking the subsequent node variables true of false starting from the root, one ends up in the formula being true or false.

The same is provable by switching to a DAG representation, where sub-formulas are shared and reorganized to avoid redundancies: if a path reaches the bottom node representing $True$, the choice of variables determined by the *if-then-else* semantics again *causes* the formula to evaluate to true. Nicely it can be proven that, given a BDD with r as a root, the set of $\subseteq$-minimal solutions — that is the $\subseteq$-minimal subsets of hook variables that need to be assigned true in order to make the corresponding path end up in the $True$ node at the bottom level — is computable, recursively, by taking:

1. the minimal solutions of the *else* branch of the node, without r (by semantics of *else*)
2. the minimal solutions of the *if* branch that are not solutions of the else branch, augmented with the root r (by semantics of if)

This is presented in figure 4.2. Suitably defining the base cases of the recursion, minimal solutions are thus readily found. We refer to the original paper for a more detailed and accurate description of the algorithm and its correctness. For as much as our interest is concerned, XSAP utilizes techniques similar to this to construct a fault tree of minimal cut sets, given the fault tree obtained by the symbolic reachability procedure towards the Top-Level Event, which is a fault tree not necessarily originating from minimal cut sets.

4.3 Using XSAP

This section is based, entirely, on the user manual of the tool [54], complemented by [30]. It is a summary of the main features that we will need in the next chapters to carry out our feasibility assessment and has no pretension of being complete. We recommend the pointed references for a comprehensive treatment.

4.3.1 Modeling with nuXmv

Prior to everything, XSAP needs a Kripke structure to work upon. As previously mentioned, this is handed straight over to the nuXmv model builder, which relies

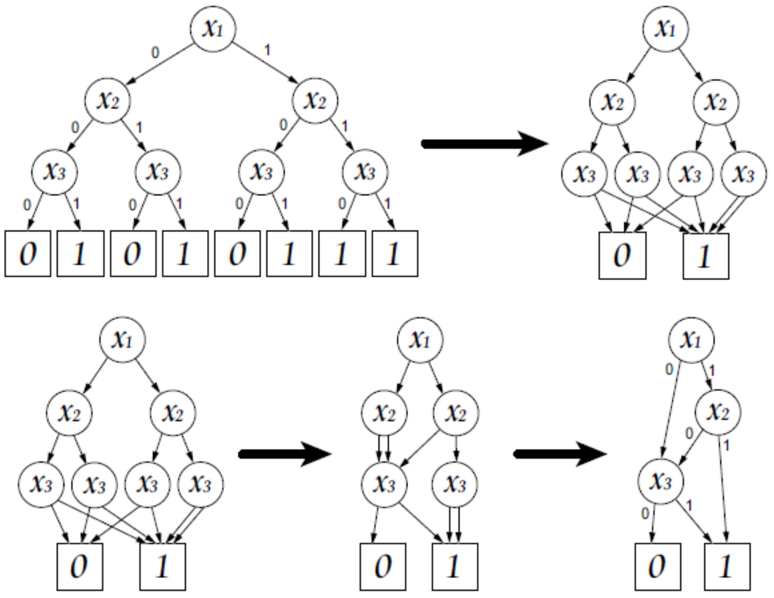


Figure 4.2: An example of BDD tree reduction. Branches labeled with 0 or 1 indicate the *else* and *if* branch, respectively. The last DAG shows cut sets very clearly, namely $\{x_1, x_2\}$, $\{x_1, x_3\}$ and $\{x_3\}$. The minimization algorithm would remove the middle cut set because not minimal.
Image originally found in http://sodans.usata.org/www.epics.jp/mc/modelchecking.html

on the SMV input language. It will be matter of the first segment of this section to elaborate on this. As a natural continuation, then, in the second part of the section we will develop a simple yet meaningful example, that will bring to light all the input language's interesting features.

Input language

The input language of nuXmv is fully detailed in the user manual[18]. We rather propose a more informal discussion in which we describe the language by use examples. This will allow us to keep the presentation simple while uncovering all the specifics needed later in chapter 6.

The SMV language has variables, that can be of three different types:

- ***state variables*** **(VAR)**: these are variables that generate the state space of the system
- ***input variables*** **(IVAR)**: these are variables that come unpredictably from the surrounding environment of the model, without possibility of control from the model (they get random values in their domain).
- ***frozen variables*** **(FROZENVAR)**: these are model variables that once assigned, they never change. They are not like constants in ordinary pro-

gramming languages, because their initial value can be absolutely random (and, in general, this is the case).

In short, state variables represent the actual variables of a system, input variables represent the non-deterministic events that might occur on the system and frozen variables represent undefined variables that are stable once their value is decided.

The domains of variables are defined by their types. In nuXmv we have *booleans*, *enumeratives*, *bounded integers* (aka *interval ranges*, such as 10..15), *word types* (aka *bitvectors*, that have the semantics of the hardware integer representation systems), *Integers* and *Reals*[7]. All together with their domains and wit a concern to IVARs, variables constitute the state space of the Kripke structure.

In order to specify the initial states and the transition relation, nuXmv provides two options: one functional and one relational. Functional relations are expressed using variable assignment, in the scope of an **ASSIGN** keyword, as:

- **ASSIGN init**(⟨var⟩) := ⟨expr⟩, which specifies the initial value of ⟨var⟩
- **ASSIGN next**(⟨var⟩) := ⟨expr⟩, which specifies the value of ⟨var⟩ in the next step of evolution, i.e. the value upon transition
- **ASSIGN** ⟨var⟩ := ⟨expr⟩, that accounts for both the initial and transition value of variable ⟨var⟩

On the other hand, the relational option expresses initial states and evolution using boolean propositions, using the keywords **INIT**, **TRANS** and **INVAR**, where:

- **INIT** ⟨expr⟩ specifies an initial constraint
- **TRANS** ⟨expr⟩ specifies a transition constraint
- **INVAR** ⟨expr⟩ specifies an invariant constraint that holds throughout evolution.

Choosing one way of encoding the Kripke structure or the other can have an impact on performance — the functional option, for instance, would be more natural for induction. Importantly, if the specified transitions are not strict enough, the variable may get any value allowed by its domain, randomly.

Expression (⟨expr⟩) can be arbitrary complex in general and they may involve:

- classical arithmetical operations such as $+, -, \times, /, mod$, for numerical values
- boolean relations such as $<$, $\geq$, $=$, etc, connected by boolean connectives, whenever booleans are involved
- operations of left and right shift (`<<`, `>>`), bit selection ($[n : l]$) and concatenation ($w_1 :: w_2$), in case of bitvectors.

Strict typing rules are forced by nuXmv. Finally expressions admit conditionals, either expressed as switches or in *if-the-else* form:

[7]Optionally, it is also possible to specify arrays of types, albeit this is equivalent to having several separate variables not being dynamic indexing allowed by nuXmv.

```
– case
    <cond_1> : <expr_1>;
    <cond_2> : <expr_2>;
              ...
    <cond_n> : <expr_n>;
    TRUE : default_expr;
  esac;

– (<condition>) ? (<truebranch_expression>)
                 : (<falsebranch_expression>)
```

If one conditional branch is taken all the following will not be considered, but overall they are required altogether to cover the domain exhaustively.

The archery example

In order to illustrate the language, we will go through a simple intuitive example where we imagine an archery player at the Olympics games.

The archer has to confront herself against the environment conditions, such as the air temperature during the performance and the elasticity of the bow string. Moreover, she has to handle the pressure of the game, which increases as the turns go by. These mentioned characteristics will be modeled as the state variables of our system, indicating, for each configuration, a different condition that the system is in. During the game session external events may happen. These could be a bothering ray of light hitting on the player's eyes or a sudden wind gust of unknown intensity, both representable in the model by using IVARs. Finally, the number of arrows to be shot in each turn and the distance to the target are frozen variables, decided in fact before the beginning of the session.

This will be all we will need for our illustration purposes. The SMV model of the system is shown in figure 4.3, which shows all the variables involved together with their initial specifications and transitions. The choice of typing the variables with discrete types is arbitrary.

All the code is contained in a **MODULE main** scope. Each SMV file needs to have one **main** module in it, in order to hook up with the SMV engine. Each file may optionally be provided with several modules, each accepting inputs for the connection with other modules. In that case modules are to be instantiated in the variable section of another existing module, such as the **main** module[8].

The initial states of the system are specified by the **init** assignments and also partially by the **INVAR** condition on the available arrows for the game. The way the pressure load is specified indicates non deterministic assignment, meaning that

[8]The idea here is that different modules represent different system components. Although separation of modules is not present in our simple example of figure 4.3, we will extensively use this concept when describing our use cases of chapter 5. We believe that this will be intuitive enough for the reader to follow.

```
MODULE main
VAR pressure_load : 0..100;
    heat : -50..50;
    bow_elasticity : {low, normal, high};
    turn : 1..10;
IVAR wind_intensity : integer;
      ray_of_light : boolean;
FROZENVAR target_distance : 5..60;
          arrows_to_shoot : {UNDEFINED, 5, 10, 15,
                             20, 25, MORE};

ASSIGN
       init(pressure_load)   := {0, 5, 10};
       init(arrows_to_shoot) := UNDEFINED;
       init(turn) := 1;
       init(bow_elasticity) := normal;

       next(turn) := (turn < 10) ? turn + 1 : 10;
       next(bow_elasticity)  :=
           case
             heat < -10 : low;
             heat > 10  : high;
            TRUE : normal;
           esac;

TRANS next(pressure_load) = next(turn) * 9
    | next(pressure_load) = next(turn) * 10 ;
TRANS ray_of_light -> next(pressure_load) > pressure_load;
INVAR arrows_to_shoot != MORE;
```

Figure 4.3: The archery contestant: conditioned feelings in an open environment

at the initial state the system is allowed to be either in a state where the variable `arrows_to_shoot` is either 0, 5 or 10. Fictionally, this would mirror the initial feeling of the contestant in respect to the game.

The transition relation of the system is specified by the **next** assignments and by the **TRANS** and **INVAR** constraints. The turn increases at every time step up to 10, which is the maximum. This denotes that session turns are evolution step. The elasticity of the string is made dependent to the air temperature, which is given by the heat. The heat itself is unconstraind, therefore it may get any value from its domain, arbitrarily in time. Finally, the pressure load is constrained in two points, in the **TRANS** constraints. The first constraint tells that the pressure load for the player increases as the end of the session approaches, linearly by a factor of 9 or 10, non-deterministically. The other constraint says that if the player gets distracted by the ray of light, its pressure for the game will certainly rise. To conclude, notice that wind intensity would have no impact in this model and neither would the distance from the target.

4.3.2 Model Extension

This section completes the description on the usage of XSAP for the part which concerns the actual construction of the safety artifacts. Very high-level, XSAP works on a nominal model, that we call the *.smv file* and a injection instruction file, called *.fei file* (Fault Injection Instruction). The instruction file begins with the following line:

```
FAULT EXTENSION <name>
```

After that, module extensions can be specified in the following lines, as short sections. The extension of a generic module is specified as follows:

```
EXTENSION OF MODULE <module_name>
    <injection_specification_with_slices>
```

Every module extension is defined by a number of *slices*. Slices are concurrent entities that modify the module's state variables. Every slice can be seen as a single act of possible injections, that together determine the overall extension of the module they refer to (hence the name).

Defined by means of mutually exclusive modes, that we will discuss in a moment, each slice must specify which state variables it is bound to act on. Only state variables can be subject of injections:

```
SLICE <name> AFFECTS <
    affected_state_variable>
   WITH
   <...modes...>
```

Modes define the options for a slice to follow. They are mutually exclusive and are to be intended as statements over a fault slice to behave in one way or another based on the mode the system is in. For example, in a first *degraded* mode (arbitrary name) the system might malfunction and the slice would perform a certain unwanted action on the variable affected by the slice. We might think of a car slowing down (action on the state variable *velocity*) and consuming more energy (action on the state variable *energy*) due to a tyre deflation in this case. In the same slice we might additionally conceive a *breakdown* mode (another arbitrary name), where a more severe contingency affects the system. We might think of a car halting due to a sudden tyre burst, which makes the tyre flat and a consequent immediate decrease of the supplied energy to 0.

Modes are specified referring to behaviours based on externally specified state machines. The platform allows for specifying this kind of machines arbitrarily. This is a major strength of the tool that makes it extremely versatile. We do not need this type of control for what concerns our explanation purposes, so we will stick, only, to the machine derived from a *stuck-at* fault semantic, which is the only type we will need in the second part to explain the concepts of this thesis.

We distinguish two variants: *permanent* and *transient*. We call those *local dynamics*[9], we call the name of the fault relative to the specific machine an *effect mode*. These names are taken from the user manual of the tool, to which we refer for a more comprehensive view of the platform.

The state machine for any effect mode is composed of two states, named *nominal* and *faulty*. There are two transitions: one going from *nominal* to *faulty*, specifying the effect mode's entering condition and one self-reentrant in the *faulty* state, specifying the effect of staying in fault mode. For the *stuck-at* effect mode this amounts to entering with a predetermined value and keeping that value while the system is faulty.

The local dynamics is expressed by another state machine, that moves between the two states of *nominal* and *fault* driven by events. Events are governed by input variables *from the nominal model*. This means that the nominal model *decides* through input variables when one mode should fail, or get fixed, or do whatever the local dynamics asks it for. These are the only point of intersection between the nominal model and the `.fei` file. We will refer to those input variable as *hook variables* in the following, because they specify how fault machines can hook to the nominal mode. In the example of a transient fault, the nominal model should have two input variables, one for the occurrence of failure and one for the fix of the fault that recovers the system in nominal conditions. For a permanent fault, the fix event would be clearly aimless[10].

The specification of the transient *stuck-at* fault is expressed as follows:

```
MODE <name>: Transient
    StuckAtByValue_I (
    data      term << <
        value_to_ba_stuck-at>,
    data      input << <needed_var>,
    data      varout >> <affected_var>,
   template   self_fix = fixed ,
    event     failure >> <hook_failure
        >,
    event     fixed >> <hook_fix>
);
```

For the *stuck-at* fault that we present, the `<needed_var>` variable and the `<affected_var>` variables are to be assigned to the same state variable of the nominal model.

[9]This is opposed to the *global dynamics* between fault modes, that we do not see here.

[10]There is a difference between the concept of an *event* and that of *template*, subtle enough to be omitted. A note was worth the mention because both words will appear in the `.fei` files of chapter 6 and below in the examples of this section.

The final skeleton for a `.fei` instruction file, given `go_faulty` and `back_nominal` as hook variables to affect variable `var_value` with a transient *stuck-at* fault, would look like the following:

```
FAULT EXTENSION FE_SIMPLE
  EXTENSION OF MODULE Simple
      SLICE slice1 AFFECTS var_value WITH
        MODE stuckmode0 : Transient StuckAtByValue_I (
                    data      term << 0,
                    data      input << var_value,
                    data      varout >> var_value,
                  template    self_fix = fixed,
                    event     failure >> go_faulty,
                    event     fixed >> back_nominal
                  );
        MODE stuckmode-1 : Transient StuckAtByValue_I (
                    data      term << -1,
                    data      input << var_value,
                    data      varout >> var_value,
                  template    self_fix = fixed,
                    event     failure >> go_faulty,
                    event     fixed >> back_nominal
                  );
         ...
         ...
         ...
```

The platform comes with useful python scripts to run the analysis at no cost. If `TLE_negated` is an invariant property of the nominal file, the sequence of calls is performed as follows (variants are not considered here):

`python extend_model.py -v <nominal>.smv <injection>.fei`
`python compute_ft.py -v --prop-name TLE_negated` Their meaning should be

clear at this point. Options are available for both scripts. For `compute_ft.py` the interesting ones are the following:

- `--engine`: allows to pick one among `bdd`, `bmc`, `ic3` and `msat`. The relative algorithms described in section 4.2.2 are used correspondingly.
- `-k`: specifies the bound for the bmc procedures (`bmc` and `msat`)
- `--gen-traces`: generates traces relative to the found cut sets of the tree. This allows to determine the ordered sequence of faults that have to trigger in order to reach the Top-Level Events corresponding to a minimal cut set.

The limited number of commands needed to run a safety analysis using XSAP is remarkable. At this point should be palpable the extreme ease and non-intrusiveness that is permitted by using the utilities of the platform. One shortcoming of the tool is that is cannot work over infinite-state systems yet, although an extension is programmed by the authors for the future [54]. The significant datum is that XSAP has over a decade development history, starting from its “ancestor” FSAP/NuSMV-SA [22]. Moreover its integration with a cutting-edge tool such as the nuXmv model-checker makes it a valuable choice for both academics and industry.

In the next chapters of the second part of the thesis we will see it in action over simple use case models of SOA-based Cyber-Physical System taken from the literature and have the actual evidences that corroborate this impression.

Part II

Novel Contributions

The state of the art: where are we?

In the first part of the thesis we discussed background knowledge and state-of-the-art on the problem we are touching on. Here and there we gave hints about the suitability of modeling and algorithmic techniques in relation to our study case. This section is introductory to the second part of the thesis and gathers the notions that we developed so far to give a collective view. In order to make the presentation of our novel contributions smoother, we briefly reconsider the tool availabilities presented in the first part of the thesis (both theoretical, such as contracts, and effective, namely XSAP) to direct how the various pieces can be fit together seamlessly in the next chapters.

Our work can be positioned at the intersection of many areas that we presented in the first part. The domain is that of Cyber-Physical Systems deployed over Service-Oriented Architectures. The literature, especially the part involving verification and formal techniques, either keeps them in separate tracks or overlooks one of the aspects. In the following part of the thesis we elucidate on our perception of the problem, hopefully clearing up on the topic that considering both aspects simultaneously can be beneficial for generality. The reason is the simplest: SOA is finding application to the Cyber-Physical System domain because it is the most natural choice for addressing easy distribution of components in a non-localized area, modeling and verification techniques of both should meet to bolster on this rationale.

Our considerations are the outcome of a thorough literature survey, that in more than one case turned out to present similar conceptions for one domain or the other. The most prominent example is on the modeling of services in SOA, that we saw is outstandingly close to the concept of contracts developed for Cyber-Physical Systems (cf. chapter 3). Moreover, the fact of treating our systems in the context of SOA traces an explicit relation with the concepts of loose-coupled distribution of work, temporality, network structure and controlled system interactions, all of which are representative aspects of SOA that are or will soon be part of the cyber-physical infrastructures. Having industry as a target of our study we are particularly perceptive to this operational aspect.

In the next chapters we will demonstrate that keeping an inclusive view of the two aspects can be gainful for research. As an example we will show in section 6.1.1 that the analysis of dependability in cyber-physical systems can benefit from a separation between functionality and network topology, which is a concept advocated, for example, in [61]. From that we also took inspiration to consider two levels of architecture, namely the functional level comprised of system dynamics and the architectural level, concerned with changes in topology.

For the study of system dependability we will use model extension with automatic construction of fault trees, exploiting the attributes featured by XSAP. For us to make this possible we will have to rethink system modeling before XSAP and understand how standard engineering practice for system modeling can be braided to it. We will have to deal with the fact that XSAP, based on nuXmv, is synchronous and therefore it models evolution by definite simultaneous steps for all components. Our adaptation will need to account for dynamics beyond the synchronous case, for errors in communication and possibly for time delays. Similar issues arise for contracts, which do not normally account for stateful evolution of systems with time, especially in case of asynchronous feedback.

From here it starts our proactive involvement, starting from our recommendation of the industrial language to prefer and going right to the modeling and dependability assessment of systems, that takes into account all the previous background to develop innovative techniques, pivoting on the functionalities provided by XSAP.

Chapter 5

Use Cases

In order to carry out our methodology under context we will make use of two simple study cases and their variants. They are respectively taken from another work on SOA [73][71][72] and from an European-sponsored project named DANSE [43]. We will not merely transcribe those two models into our setting, but adapt them and render with as few unnecessary details as possible and trying to keep pivotal points that highlight our methodology, make up some if they are not present in the original work.

We will start out the chapter by surveying, shortly, the existing system-design methodologies for the modeling of SOA, among which we will pick one and later use it to model and present our use cases. The modeling language is important in order to help selecting the right level of abstraction for the models under consideration. At the end of the thesis we report a proposed methodology to transform models from the selected language to SMV files with fault extension in place. This work has not yet been automatized but it is planned for future work.

5.1 Modeling Languages

SOAs traditionally distinguish *service orchestration* to *service choreography.* Orchestration features a director telling its ensemble how and when they should perform activities; choreography is more about a prepared collaborative strategy with no central point of direction. The difference is a matter of control perspective. For example orchestration is the activity that online shops provide on the internet: upon demand for an item they orchestrate their own service, the payment service and the shipping service for accomplishment of the delivery. Modeling a system in a choreographic fashion would mean to have all components interchanging messages independently, for the accomplishment of a common goal.

There are quite some ways to describe how an SOA is structured, from an implementation viewpoint. Among the most popular languages we report WS-BPEL [58], WS-CDL [91] OWL-S [77] and WSMO [92]. The focus while designing those languages has been put on compositionality and interaction patterns. Here we refer to [10], which discusses the applicability of those languages in the context of SOA.

The first two languages (WS-BPEL and WS-CDL) are based on process execution, meaning that, as modeling languages, they can be run. Both are based on XML and are static, i.e. their behaviour, especially the composition patterns, can be known in advance and is independent from execution time events. Their difference is on the modeling criterion: while WS-BPEL is used to describe service orchestration, WS-CDL describes choreographies. We mentioned the distinction between orchestration and choreography in the introduction, as the property of having a centralized or a distributed control of the evolution.

OWL-S and WSMO are ontology-based dynamic languages, whose behaviour gets defined during execution, based on information provided at runtime. The key difference between the two languages is in what they put in the spotlight: services for the former and its own ontology the latter. OWL-S distinguishes four types of ontology, that separate their behaviour in what we would call, inspired by our contract basis of chapter 3, viewpoints. These comprise the functional and the non-functional view and can be expressed by a choice among three ontology expression languages, namely SWRL, DRS and KIF. Neither of them has a clearly understood companion semantics. Because of the variability of languages for expressing logical formulas, OWL-S can be adapted both to describe systems based on orchestration and choreography. We refer to [65] for a neat and deeper comparison between the two languages.

We are not interested in process execution modeling languages, because our work has more abstract applicability than implementation. However, understanding the features of those languages lets us understand the principles of SOAs, as a reflection of their constructs: the distinction between orchestration and choreography, the need for complex interaction patterns and the separation between timing and functional viewpoint are some of these significant features. Importantly, their co-existence is not accidental and neither it is competitive: they are different solution languages for the implementation of SOA under different, maybe intersecting, scenarios.

Ultimately, we will need means for the modeling of SOAs, free of all domain-dependent details. We will need a design language for only capturing the most effective traits of the architecture, inclined toward verification. BPMN can be cast as such a choice [25]. It provides a graphical language to specify service interactions as workflow activity diagrams, similar to UML's, but with a clearer semantics. Although appropriate for the description of web services, BPMN finds hindrances when it comes to face cyber-physical dynamics.

More common, especially in industry, is the adoption of UML with its extensions, one notable being SysML [85]. We assume here a basic knowledge about the UML language by the reader and hint at the features introduced by SysML. The central feature is the introduction of system views for the modeling of physical components' interactions. The design is subdivided into design of structure, design of behaviour and requirements specification. These are supported, respectively, by the following:

1. <u>Structure</u>, comprising *Package Diagram*, the *Block Definition Diagram*, the

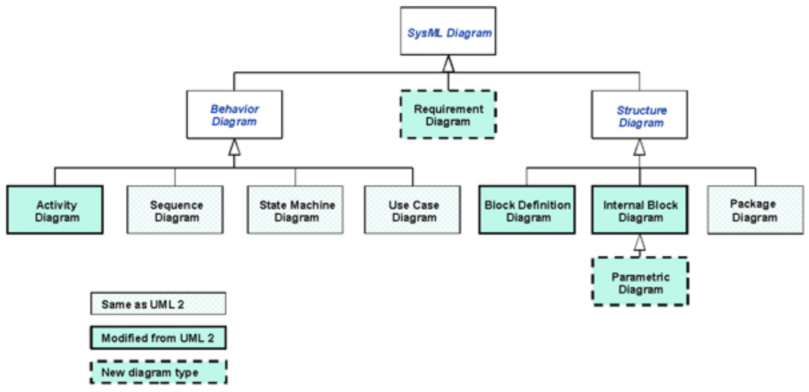


Figure 5.1: Diagram types for SysML image taken from `http://www.omgsysml.org/`

Internal Block Diagram and the *Parametric Diagram*;

2. Behaviour, comprising the *Activity Diagram*, the *Sequence Diagram*, the *State Machine Diagram* and the *Use Case Diagram*

3. Requirements, comprising the *Requirements Diagram*.

Figure 5.1 gives the graphical representation of this diagram type hierarchy.

Most of the diagrams are taken straight from UML, others are lightly modified to system engineering. For example, block diagrams are the equivalent of UML class diagrams tweaked to describe system components, or internal block diagrams are another facet of UML's composite structure diagrams where the component is described at its functional level, introducing flow ports. The parametric and requirements diagrams are novel and are used, respectively, to indicate parametric constraints between structural elements and functional/non-functional requirements on system components. Please refer to [85] for a more complete description of SysML features.

In SOA design, it is not enough to rely on the features described above, because there are parts, such as the system architecture, that need to be treated with specialized care. Aware of these matters, the research community have proposed some adaptations or extensions of UML [81][81], neither of which standardized and thus growing no appeal to general industry.

Exception to this rule is given by SoaML, OMG standard [83]. SoaML makes the following diagrams available[1]

- *Service Interface diagram*: it defines the programmatic interfaces used by participants.
- *Participants Diagram*: it describes the behaviour of a single service, or participant, using ports at the interface level. They can be receptive ports (*request*

[1]The complete description of them is beyond the scope of the thesis. Refer to [83][84] for a wider view on the topic.

ports), service provision ports (*service ports*) and declare the use of specific service interfaces.

- *Service Contract Diagram*: it defines the SoaML service contracts, or *s-contracts*[2]. S-contracts are single interactions patterns connecting participants' interface ports. Any number of participants can partake any s-contract.
- *Service Architecture Diagram*: it defines the architectural interactions between participants through the modeling of roles participating in s-contracts.
- *Capability diagram*: it defines the relations between services using the ≪*use*≫ stereotype.
- *Message Diagram*: it defines the type of exchanged messages

The core of SoaML is on the definition of the Service Architecture Diagram. Let us suppose that an s-contract defines a costumer-retailer single interaction. The s-contract will be defined on the two linked roles of server producer and requesting consumer. We would have two participant diagrams, costumer and retailer, that expose a request port and a service port, respectively. Finally the two types of diagrams would be combined into the service architecture diagram, that links the costumer participant to the s-contract in the role of consumer, while the retailer participant to the role of producer. The link between participants and roles is given by the service interface diagram. More complex architectures can be defined, possibly involving more than one s-contract and more than two participants.

SoaML has been designed for the definition of web services, but it has retained not much domain specificity from there. The view is on constructing the service architecture, using interface relations between participants. Using SysML for the functional and requirement part of the specification we can combine the two modeling languages and obtain the separation between functionality and architecture that we previously advocated in section 2.2. This is good for other reasons, mainly related to industry needs and corresponding engineering's familiarity with UML and their derivations. But also notice that the languages are both standard and not domain-specific. They have disjoint non-conflicting types of diagrams that thus do not lead to conflicts. Finally they can optionally be used each one alone, whenever use case descriptions do not need the other.

Let us try to explicate what is the gist underlying our just-made assertions: let us ask ourselves why could we not use alternative languages such as AADL, for example [1]. After a short analysis one can easily find out that AADL is rather complicated for the average engineer, whereas she would have more confidence with UML and derivative languages. Moreover, as mentioned in section 2.3.3, AADL does not have support for services and the complex fault model it provides would only be of little use for us, because faults are trivial in most of the cases that we

[2]Alas, contracts. Please notice that this kind of contracts have nothing to do with the contracts of chapter 3. It is for convenience that we distinguish the two concepts calling SoaML contracts *s-contracts*.

are willing to consider[3].

The combination of SysML and SoaML, that we will hereafter refer to as SysML+SoaML, seems to be the perfect fit to our needs. We would like to remark, however, that most of the novelty presented in this thesis is independent from the particular language employed: under similar conditions and with a little bit of massaging, the same techniques could be recovered for other kinds of modeling languages.

5.2 The simplified CAE example

The first use case that we present is a simplified version of the CAE (Concept Alignment Example) emergency response system, taken from the DANSE european project deliverable D3.3 [44] and representative of a number of study cases aimed at the organization of safety plans for the associative recovery from alarming situations. Dependability on those structures is in most cases critical for people safety.

The reason why we chose this example was to specifically model the evolutionary development of SOA-based systems in terms of participants arbitrarily leaving the playboard. This is not new to the literature as already present, e.g., in [57] and [56]. The CAE system that we will present is a simplified version that highlights the key features of our approach. The scenario is that of a urban area, a *District*, that relies on a *Fire Station* to prevent unexpected fire explosions to burn over. The Fire Station has a number of *Fire Fighting Cars* available, 5 in our specific case, that it can send to mitigate the fire. We assume that one car is always enough to the district in order to mitigate one fire explosion.

The SOA structure is that of the 7 participants (the District, the Station and the 5 Cars) communicating with each other according to agreed modes (see figure 5.2). Upon fire, the District sends a help request to the Fire Station. The Fire Station immediately gathers its available resources and sends a signal to one of its Cars to reach the place. At that point the Car readily moves on the district, extinguishes fire and sends an acknowledgement to the district. Then it goes back to the Fire Station sending a message of mission accomplished. This makes the Fire Station acknowledge that the car is back operative once it gets to the station. We assume that a car needs to go back to the station whenever it quenches one fireburst (to refuel, alternate firemen, ...). This emergency response process reiterates whenever needed against any new fire explosions upon service unavailability (i.e. when no more cars are available to send).
The activity diagram of the CAE for one car is presented in figure 5.3.

Virtually we could back the process by an actual SOA interaction pattern whose shallow communications are only the eventual service bindings. For example, upon fire, the District, personified by a district inhabitant, might call an emergency number and ask for a service that could extinguish the fire. The emergency number will

[3]Potentially, if this was not the case, we could construct the model design with our UML derivatives and ask interactively to the user which faults to insert, to be chosen from a limited fixed library of a few. Notice that this approach would resemble the work in [23]

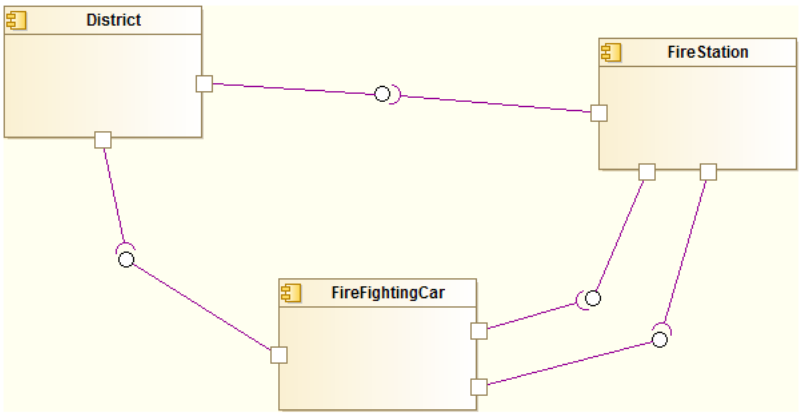


Figure 5.2: The CAE interaction block diagram.

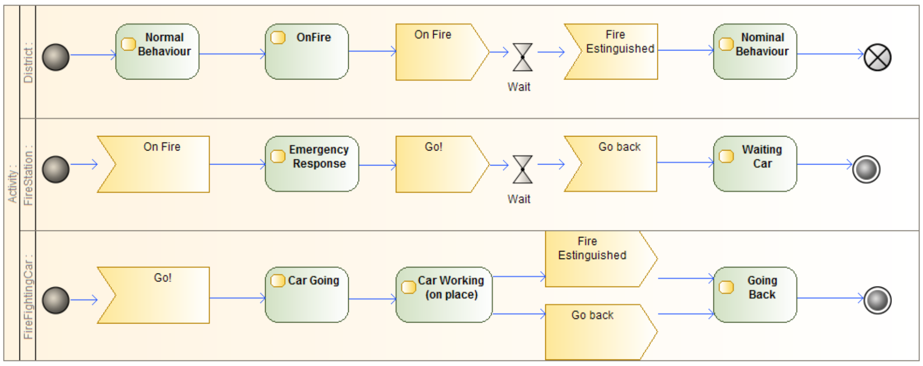


Figure 5.3: The CAE activity diagram for one car.

ensure that the citizen can tell the correct address and problem, then it redirects the call to the district's Fire Station, which after understanding the problem and location sends signals to the Cars. Signals to the cars could as well be treated in a service-oriented fashion, by backing on a service that arranges cars according to availability and as a matter of fact this kind of reasoning can be carried out for all communications and is left to the imagination of the reader.
We decided not to model all those details for ease of exposition and to keep our use case as simple as possible.

5.3 The SOA based Thermostat

In this section we will present a simple use case that we took from [71] and adapted to our needs. It is about a SOA representation of a thermostat control system, capable of regulating itself by globally choreographing services provided by a sensor, a controller and an control actuator device.

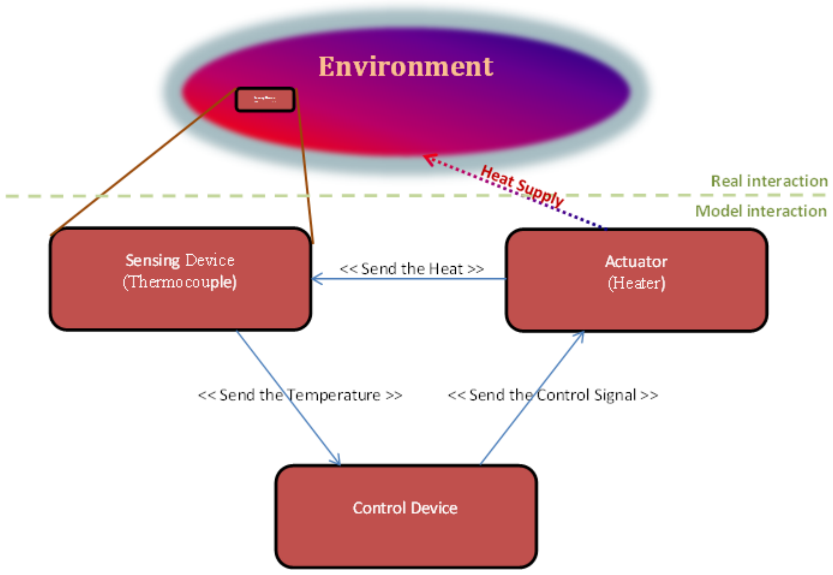


Figure 5.4: The SOA thermostat interaction model. The real heat supply and sensing of the temperature is abstracted in the model by communicating the heat supply and computation of the temperature.

We first give an overview of the system and its components, plus an intuition about the differences with respect to [71] and new goals. After that we give a textual specification of the system with an interface contractual model and define the fault modalities that the system is subject to.

At the end of the section we provide the SysML+SoaML system design of the use case.

5.3.1 System Description

The thermostat system is composed of three participants: a Thermocouple that senses the heat and translates it to temperature, a Controller that sends a control signal to the actuator given the temperature sensed by the Thermocouple and the Heater that actuates the heat supply according to the control signal. As in the source paper, we fictionally assume that the temperature is not directly sensed by the thermocouple from the environment, but instead computed by the thermocouple, internally, after it gets the heat supply from the actuator. In other words the real phase of providing heat to the environment and sensing it back to the system through the thermocouple is abstracted by a SOA communication between the actuator and the thermocouple and internal computation of the latter. The system is depicted in figure 5.4.

The work in [71] presents a technique for the injection of faults in the system at the network level, that modifyes the XML formatted messages of SOAP using a unwrap-modify-rewrap procedure at the endpoints of the communication. For their use case, the authors are interested in studying the latencies that the network is subject to, under these injected faults. Their work is interested in software fault injection (network fault injection to be fair) rather than model extension.

Although applying at different points in the design process, both our approach and that in [71] aim at the system dependability assessment; as a consequence, their needs are similar to ours. We will try to be more comprehensive though, presenting an extensive treatment of system requirements. Not all of them we plan to verify, however an extensive list shall be guidance of our work and exposition of our needs.

5.3.2 System Specification

The thermostat example is valuable because of its simplicity, allowing us to pursue our methodological approach bare of all intricacies. Still it is close enough to the Cyber-Physical world and not subject to the high dynamicity typical of SOAs. The example is also available for extensions, by either refining the single system components (introducing redundancies for example) or enlarging the whole system by introduction of new components (such as a new thermocouple placed somewhere, distant from the other). Possibly we could even add a human supervisor, as suggested in [71], that regulates a new reference level of temperature by communicating it to the thermostat system using a finite timed machine, or maybe add sudden unexpected events specific to the application, such as the opening of a window in a room making the temperature instantly drop. We shall extend our methodology further towards the dynamicity aspects of SOAs only later, when the static features will be all covered.

For the sake of fiction, we assume that the environment embedding/needing the devices is a space surrounded by walls, where the colder external ambient is big enough to neglect any temperature variation due to heat transfer. We will call this space a room – which has a tangible intuition of a specific structure – and will parameterize it as if it were a room in a building, but it could equally well be a biological cell in a liquid (with some fanciful way of transfering heat to it) or a sphere full of gas whose interest in keeping a hot temperature comes from a physical experiment. It is beyond the scope of our work to present a sound and accurate model of the system, so we will just present one that can be taken as input for our analysis and features the relevant aspects of the SOA interactions.

We thus outline in the next short sections a number of desirable functional and non functional properties on the thermostat architecture, that altogether provide the system specification. For the sake of completeness we also report, in figure 5.5 and following, the SysML+SoaML description of the Thermostat use case. The parameterization of the systems will be explicitly specified in section 5.3.4.

Remark 1 (A note on participant's interactions). Every communication is characterized by two participants, having the receiver put a service request on the channel and the sender accordingly reply to it with the value information, thus fulfilling the service. This point reveals, undeniable, the lack of dynamicity that the system presents, mentioned at the beginning of the section. Our simple use case features

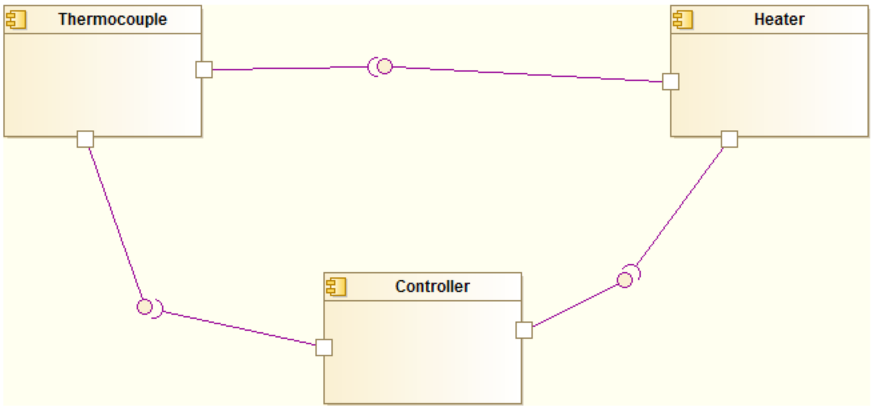


Figure 5.5: The SysML+SoaML description of the Thermostat use case - Block Diagram

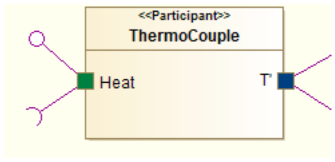


Figure 5.6: Sensor Participant

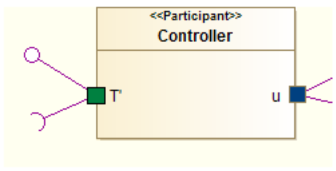


Figure 5.7: Controller Participant

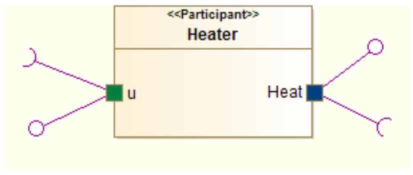


Figure 5.8: Heater Participant

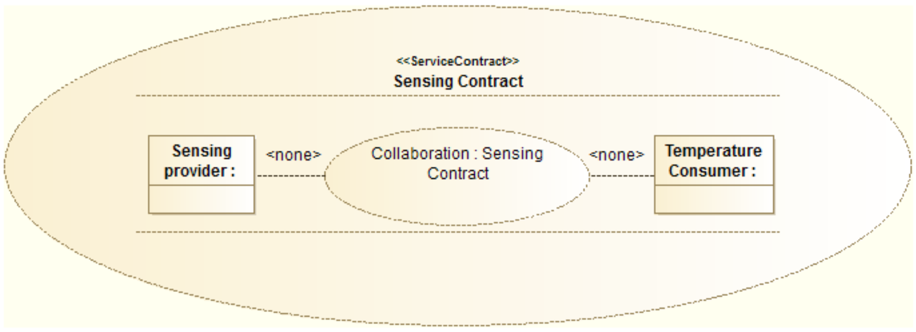


Figure 5.9: The SysML+SoaML description of the Thermostat use case - Sensing Contract Diagram

in fact, over and over, the exact same message exchanges, between the same participants, requesting the same services and under a persistent kind of binding. All those are rigid conditions considering the more typical SOA scenario where a dedicated third party is engaged every time a service is needed, where there is no guarantee of participants (or services more generally) to endure and where not as

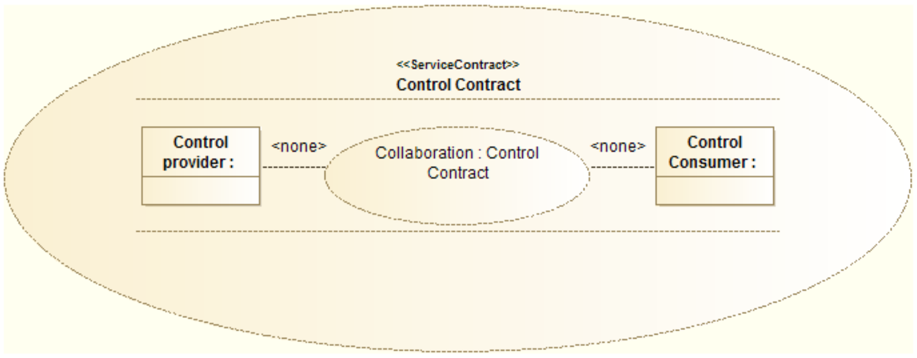


Figure 5.10: The SysML+SoaML description of the Thermostat use case - Control Contract Diagram

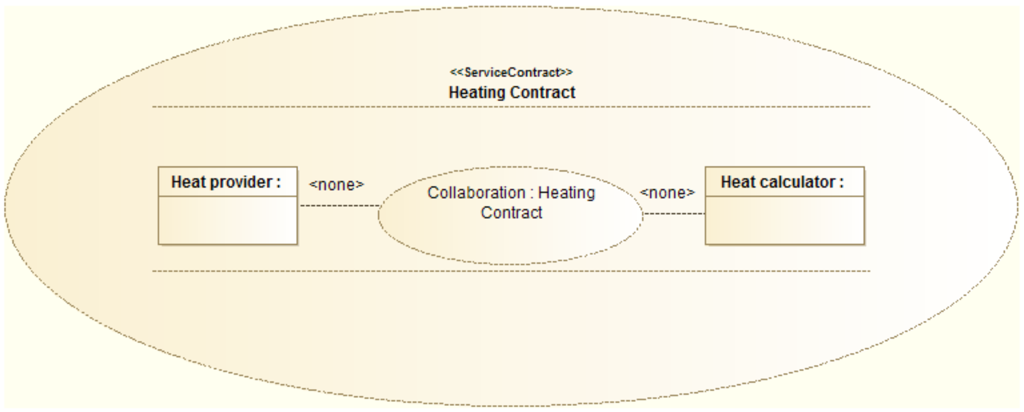


Figure 5.11: The SysML+SoaML description of the Thermostat use case - Heating Contract Diagram

much procedural and well-defined behaviour is to be expected. For this phase of our study, however, we will stick to this option and engage in more dynamicity later. This will allow us to unfold the analysis on the (simpler and static) SOA based Cyber-Physical setting with an involvement of control, which is a seldom studied case in the literature.

The Thermostat

As an overall system, the thermostat is given the reference temperature as input and returns the sensing of the temperature at a regular rate. After restricting the reference temperature to lay, sensibly, between 15°C and 25°C, we require the system to always stabilize within 10 minutes.

r1 The reference temperature will be given in °C and range in $[15, 25]$

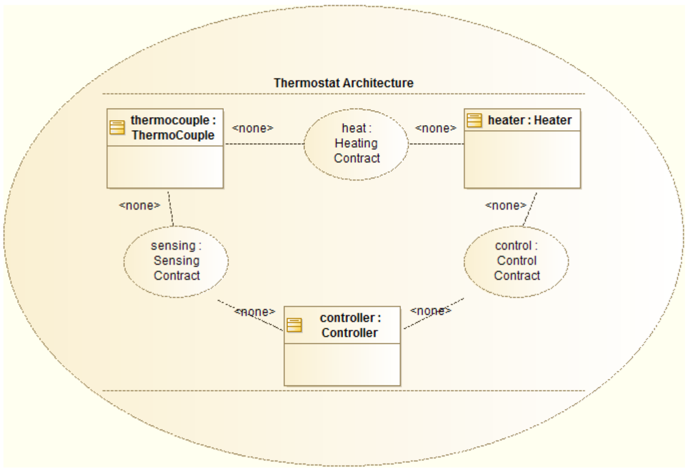


Figure 5.12: The SysML+SoaML description of the Thermostat use case - Service Architecture Diagram

r2 The system reaches the reference temperature within 10 minutes and never deviates of more than 1°C thereafter (unless the reference temperature itself is changed).

The Thermocouple

The Thermocouple senses the environment at the rate of one second. In our model convention this simply translates to saying that the Sensing Device sends the temperature no earlier than every second to the Control Device (assuming, of course, that a single measurement is never sent twice to the Controller).

For the thermocouple we require that the heat supply doesn't get too intense, case in which the thermocouple in not functional (too much heat on the sensors might lead to high internal temperatures and breakdowns). Under favorable conditions, although, the thermocouple ensures a time-proportional rise in the temperature of the room, according to how much the heat supply outdoes the maximum dissipation allowed for the room.

The maximum dissipation depends on the size of the room and its level of insulation. Based on the calculations presented in the next section (and the power of our actuator), we assume that the maximum dissipation amounts to 7500KW. Requirements r4 and r5 are particular demands on temperature increase per second.

r4 The thermocouple is able to work against power levels no higher than 70KW.

r5 If the heater supplies energy at 2100W more than the maximum dissipation (7.5KW), the temperature sensed by the thermocouple increases by a rate of 0.06°C per second. the reference temperature itself is changed).

r6 If the heater supplies energy at exactly the maximum dissipation (7.5KW) or up to 10W more, the temperature sensed by the thermocouple increases by a rate of no more than 0.0003°C per second.

The Controller

The Control Device has a constant reference temperature threshold as input, sensibly to be set between 14°C and 28°C, plus the temperature feedback from the Thermocouple that is assumed never to be lower than 0°C nor higher than 30°C. The output is given by a control signal, meant for the actuator, for the calculation of the energy value of the heat. The control signal takes values in the continuous interval $[-1, 1]$, where -1 means that the temperature should be lowered, 0 kept constant and 1 that it should be increased.

r7 The heat controller is able to bring the temperature of the room up to steady 14°C-28°C.

r8 The temperature sent to the controller is assumed never to be below zero, neither higher than 30°C

r9 The output of the heat controller ranges continuously between -1 and 1: negative values will signal the need of lowering the temperature level, positive values increasing it and 0 to keep it steady.

The Heater

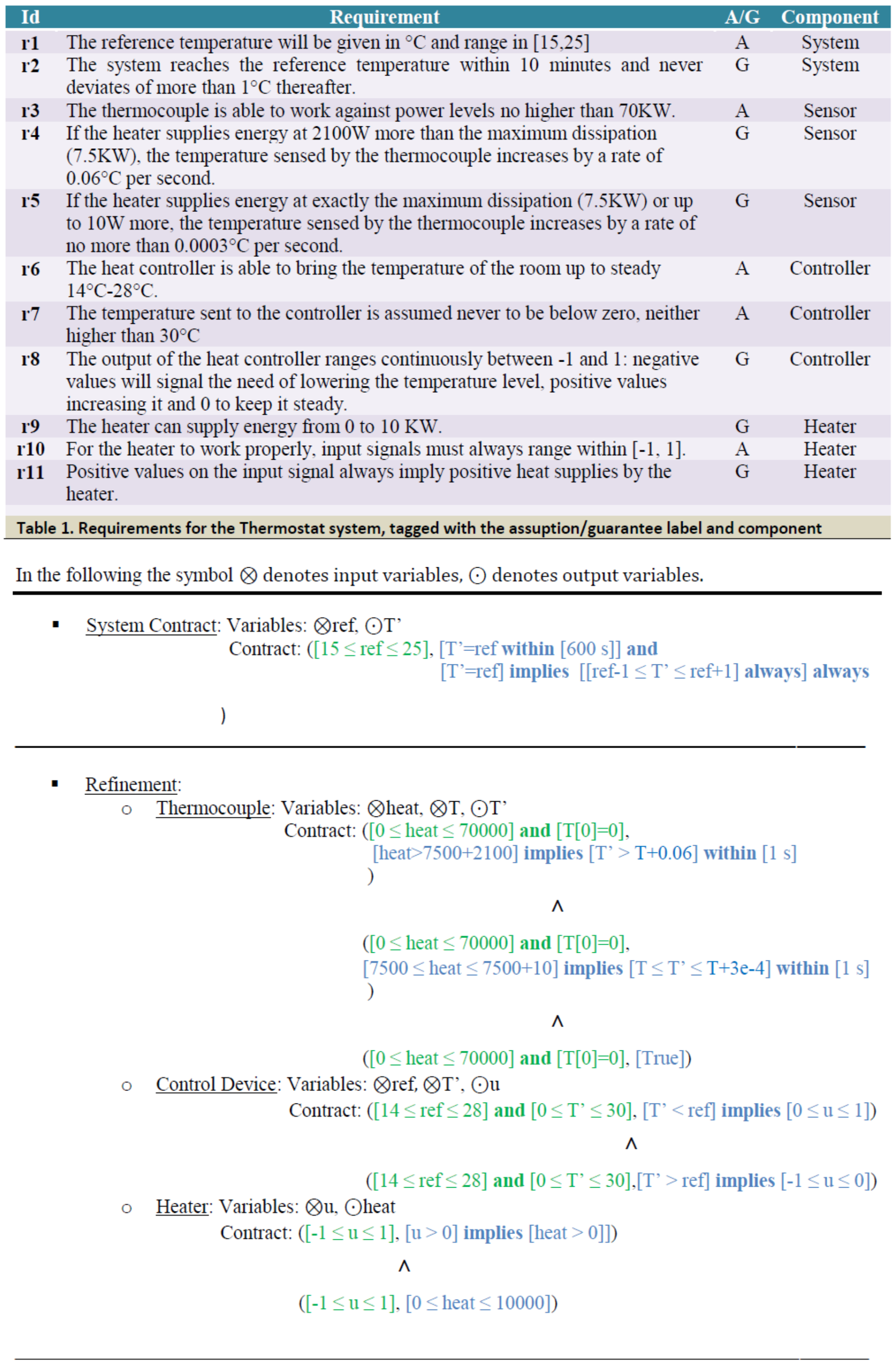
The Heater energy supply would exercise continuously in the real world, but in our model representation it couldn't, due to the latencies introduced by the SOAP transmission. For the sake of our modeling purposes, we assume that the Actuator sends a heat signal to the Sensing Device in the form of an energy derivative. This value gets summed up to the current energy value of the environment system, which should take into account energy dissipations.

The Heater can supply energy from 0 to 10 KW. The actual value is internally calculated by the Heater, accounting for the input signal, and then actuated. The input signal takes values in the continuous interval $[-1, 1]$, where -1 means that the temperature should be lowered, 0 kept constant and 1 that it should be increased. The heater ensures it will always respond with a positive heat supply to a positive input signal.

r10 The heater can supply energy from 0 to 10 KW.

r11 For the heater to work properly, input signals must always range within $[-1, 1]$.

r12 Positive values on the input signal always imply positive heat supplies by the heater.



Id	Requirement	A/G	Component
r1	The reference temperature will be given in °C and range in [15,25]	A	System
r2	The system reaches the reference temperature within 10 minutes and never deviates of more than 1°C thereafter.	G	System
r3	The thermocouple is able to work against power levels no higher than 70KW.	A	Sensor
r4	If the heater supplies energy at 2100W more than the maximum dissipation (7.5KW), the temperature sensed by the thermocouple increases by a rate of 0.06°C per second.	G	Sensor
r5	If the heater supplies energy at exactly the maximum dissipation (7.5KW) or up to 10W more, the temperature sensed by the thermocouple increases by a rate of no more than 0.0003°C per second.	G	Sensor
r6	The heat controller is able to bring the temperature of the room up to steady 14°C-28°C.	A	Controller
r7	The temperature sent to the controller is assumed never to be below zero, neither higher than 30°C	A	Controller
r8	The output of the heat controller ranges continuously between -1 and 1: negative values will signal the need of lowering the temperature level, positive values increasing it and 0 to keep it steady.	G	Controller
r9	The heater can supply energy from 0 to 10 KW.	G	Heater
r10	For the heater to work properly, input signals must always range within [-1, 1].	A	Heater
r11	Positive values on the input signal always imply positive heat supplies by the heater.	G	Heater

Table 1. Requirements for the Thermostat system, tagged with the assuption/guarantee label and component

In the following the symbol ⊗ denotes input variables, ⊙ denotes output variables.

- System Contract: Variables: ⊗ref, ⊙T'
  Contract: ([15 ≤ ref ≤ 25], [T'=ref **within** [600 s]] **and**
  [T'=ref] **implies** [[ref-1 ≤ T' ≤ ref+1] **always**] **always**
)

- Refinement:
  - Thermocouple: Variables: ⊗heat, ⊗T, ⊙T'
    Contract: ([0 ≤ heat ≤ 70000] **and** [T[0]=0],
    [heat>7500+2100] **implies** [T' > T+0.06] **within** [1 s]
)
    ∧
    ([0 ≤ heat ≤ 70000] **and** [T[0]=0],
    [7500 ≤ heat ≤ 7500+10] **implies** [T ≤ T' ≤ T+3e-4] **within** [1 s]
)
    ∧
    ([0 ≤ heat ≤ 70000] **and** [T[0]=0], [True])
  - Control Device: Variables: ⊗ref, ⊗T', ⊙u
    Contract: ([14 ≤ ref ≤ 28] **and** [0 ≤ T' ≤ 30], [T' < ref] **implies** [0 ≤ u ≤ 1])
    ∧
    ([14 ≤ ref ≤ 28] **and** [0 ≤ T' ≤ 30],[T' > ref] **implies** [-1 ≤ u ≤ 0])
  - Heater: Variables: ⊗u, ⊙heat
    Contract: ([-1 ≤ u ≤ 1], [u > 0] **implies** [heat > 0]])
    ∧
    ([-1 ≤ u ≤ 1], [0 ≤ heat ≤ 10000])

Figure 5.13: The compact specification of the Thermostat use case

5.3.3 Contracts

All participants are independent from each other, which amounts to acknowledging every interaction as asynchrounous, overlooking at the rest of the system, even ignoring its very existence. Service-based communication and independence of the participants (sometimes called loose-coupling) is an important aspect of SOA based systems, that is retained by our use case (cf. section 2.2). The contract interfaces of the system at the functional level are presented in figure 5.13 using the BCL language [52]. They formalize the requirements r1-r11 provided in the system specification that we report, compactly, in the same figure, Table 1.

Clearly the system at hand is feedback-based, thus the composition could be given as a parallel composition of each component. Here there is an hindrance preventing us to do that, namely the notion of elapsing time. As a matter of fact the Top-Level contract is expressed using a range of 600 seconds, whereas the single components are defined by timings smaller than 1 second. In order to reach the bound of 600 seconds we need a contract reiteration, persistent up to 600 steps at least. Moreover we have the notion of temperature *changing* over time, that increases according to the contract specification. Given these very sketched reasons, we acknowledge of not being able to rely on the contract framework of chapter 3 barely, because the operator of parallel composition cannot be used at support in front of such subtle trickinesses. However we can reduce to contract's parallel composition, unfolding the contract specification across scores of different contracts.

The idea is to devise one contract for each time step, knowing that the duration of one timestep is 1 second. The starting point, the *base case*, is the contract composition of the three components at time $t = 0$. Here we have a temperature of $0°$C, known from requirements. We rename the output port *heat* of the Heater as $heat_0$ and put the three contracts in parallel composition. This gives us component Thermostat_0. Using a similar trick we can rename the input port of the Thermocouple as $heat_0$ and the output port of the Heater as $heat_1$. Parallel composition of those would produce the component Thermostat_1. Reiterating the process 600 times would finally produce the whole set $\{\text{Thermostat}_i\}_{0 \leq i \leq 600}$ of interface components.

At the end, the parallel composition of these new components gives the final component that takes the reference temperature ref as an input and returns Thermostat_{600} as an output, where inputs and outputs are intended as per section 3.2. An easy algebraic analysis shows that the resulting composite contract refines the Top-Level interface contract of the Thermostat, implying that all implementations of the one are implementations of the other (cf. chapter 3). This fact will turn out to be key for scalability in section 6.1.2.

The interesting part here is that there is a notion of computation underlying the unfolding, which makes the resulting contract composite. In our case it would be an unbounded unfolding that we can end by the domain knowledge of 600 steps be enough. If we were to automate the satisfaction checking, a great opportunity would be given to us for free by exploiting the SMT-based BMC features of the nuXmv solver. Once again we stress how the reliance upon such a multi-purpose tool can come in handy in many circumstances.

Although we did not thoroughly follow a systematic computer-based refinement checking procedure for the contract of the thermostat, it is easy to speculate that automation would both be feasible and necessary, especially on models of growing complexity. We postpone the in-depth study of this aspects to future work and instead focus on the analysis and integration of existing and new techniques to understand their predisposition for the assessment of system dependability.

5.3.4 Implementation Model

We conclude the chapter by a description of the interface components that we devised in view of simulation and dependability assessment needed in next chapter. Here we describe separately the Thermocouple component, the Controller and the Heater, explaining the actual system modeling and choice of parameters.

For our use case example, the Thermocouple participant would have to calculate internally the actual dynamics of the physical system. For explanatory purposes we will calculate the energy of the system by supplying the heat provided by the Actuator to the internal energy of the system, which we will assume be decaying according to a basic heat conduction system. We take [89] as a reference for the involved equation, that we report here:

$$\frac{\mathrm{d}Q_{out}}{\mathrm{d}t} = \alpha_{cond} \cdot A \cdot (T - T_a) \tag{5.1}$$

Here Q_{out} is a time-dependent variable that measures the heat natural dissipation in homogeneous conditions, $\alpha_{cond} = [5, 20]\, W/(m^2 \cdot K)$ is the heat-transfer coefficient of the wall, $A = [50, 500]\, m^2$ is the surface of the room exposed to the outside, T is the temperature of the room and T_a is the ambient temperature facing the wall on the outside.

For simplicity we assume that once the heat is supplied, the whole system entirely gains it, homogeneously, in the delta fraction of time that the adopted step calculation is on. In this way we can add the heat to the internal energy of the room algebraically and, consequently, assume that the final temperature is homogeneous in the room. We also assume that there are no other heat dissipations than conduction through the walls and that the heat supplied by the Actuator is passed sharply and net to the Sensing Device. Moreover we require T_a constant at 0°C, so that it can be removed from the equation above. A more complex and accurate system of equations could be devised, but that is beyond our goals for the simple use case at hand.

The heat supply from the Actuator will thus contribute positively to the internal energy of the room, whereas the dissipation will be wasted energy. The new temperature of the room, T', can be found from the well known heat equation $\Delta Q = C_s \cdot m \cdot (T' - T)$ and combined with 5.1 in ΔQ_{out}, as:

$$T' = \frac{\Delta Q_{room}}{C_s \cdot m} + T = \frac{\int_{\Delta t} heat\, \mathrm{d}t - \Delta Q_{out}}{Cs \cdot m} + T = \frac{\Delta t \cdot heat - \Delta Q_{out}}{C_s \cdot m} + T \tag{5.2}$$

...where $Cs = [1000, 1015]\, J/(Kg \cdot K)$ is the [isobaric] specific heat of air and $m = [35, 400]\, kg$ is the mass of air in the room[4].

The Control Device can be implemented by a PID controller [7], whose parameters can be optimized manually, whereas the heater can be simulated by a function that transforms the input in a heat value according to its contract. We decided that the named function would have to behave according to the following formula:

$$heat = \min(heat + 200 \cdot u, 10000)$$

...starting from $heat = 0$ and provided 10000 the maximum bound of $10KW$ available to the heater and u the control signal. It will be the controller's duty to input sensible values of u for the temperature regulation. If $Err = (Tref - T)$ represents the deviation from the reference temperature, we obtain the value of the control input signal as:

$$u = K_p \cdot Err + K_i \cdot \int_{\Delta t} Err \, \mathrm{d}t + K_d \cdot \frac{\mathrm{d}Err}{\mathrm{d}t} \tag{5.3}$$

In order to understand what sensible parameters the PID controller should be tuned with, we made simulation experiments using the Matlab/SimuLink framework. The same parameters were then used for the nuXmv model of the thermostat. The top level view of the Matlab/SimuLink model is presented in figure 5.14, where the dynamics of the model are encoded into the single blocks.

We set up the following simulation parameters (that we will retain in the nuXmv model):

α_{cond}	$=$	$5W/(m^2 \cdot K)$	m	$=$	$35kg$	Δt	$=$	$1s$
C_s	$=$	$1000J/(Kg \cdot K)$	A	$=$	$50m^2$	T_{ref}	$=$	$293K$

Under those conditions, the best PID controller was found, manually, by setting the proportional and derivative gain respectively to $K_p = 0.02$ and $K_d = 1.5$ and excluding the integral component (i.e. $K_i = 0$). The resulting fault-free functional system, whose performance are reported in figure 5.15, would thus be able to satisfy the top-level contract of the thermostat specification, which asked the system to reach the reference temperature within $600s$ and stabilize. Notice that, as a result of the simulation, we did not prove that the system contract is satisfied: to do that we would have to show that a similar stabilization property is available against all reference temperatures within 15°C and 25°C. We became confident of that property being true, however, after observing that for the boundary cases (i.e. $T_{ref} = 15°C$ and $T_{ref} = 25°C$) the system would similarly stabilize before $350s$ as in figure 5.15, never exceeding the 1°C error as the contract demands. For the sake of completeness, in figure 5.16 and 5.17 we also report the time evolution of respectively the input signal u and the *heat*.

[4]Values for C_s are taken from `http://en.wikipedia.org/wiki/Heat_capacity` and approximated. The mass of air in the room is found by an application of the ideal gas law at $1atm$ pressure at a temperature level between 0°C and 30°C, assuming the size of the room between $30m^3$ and $300m^3$ and approximating the molar mass of [dry] air as $0.029kg/mol$ (plus rounding).

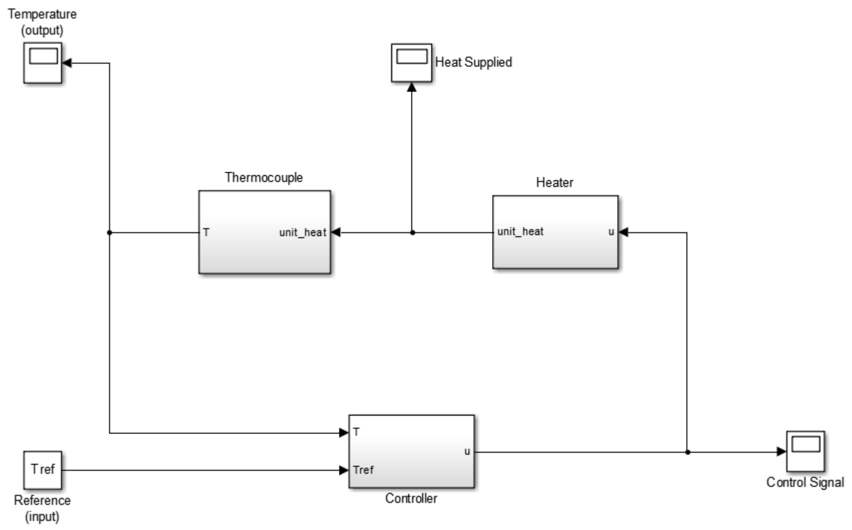


Figure 5.14: Top-level view of the Matlab/SimuLink thermostat model

Notice that the proposed Matlab/SimuLink model intentionally focuses on the nominal dynamics of a system without latencies. It was not the modeling of the system as a SOA the purpose of our analysis; it was just resolving on the needed conditions that our thermostat architecture should be bound to operate in, in order to set sensible parameters to the nuXmv model that we will will have develop in the next chapter, section 6.1. There we will see the SOA essentials going on stage.

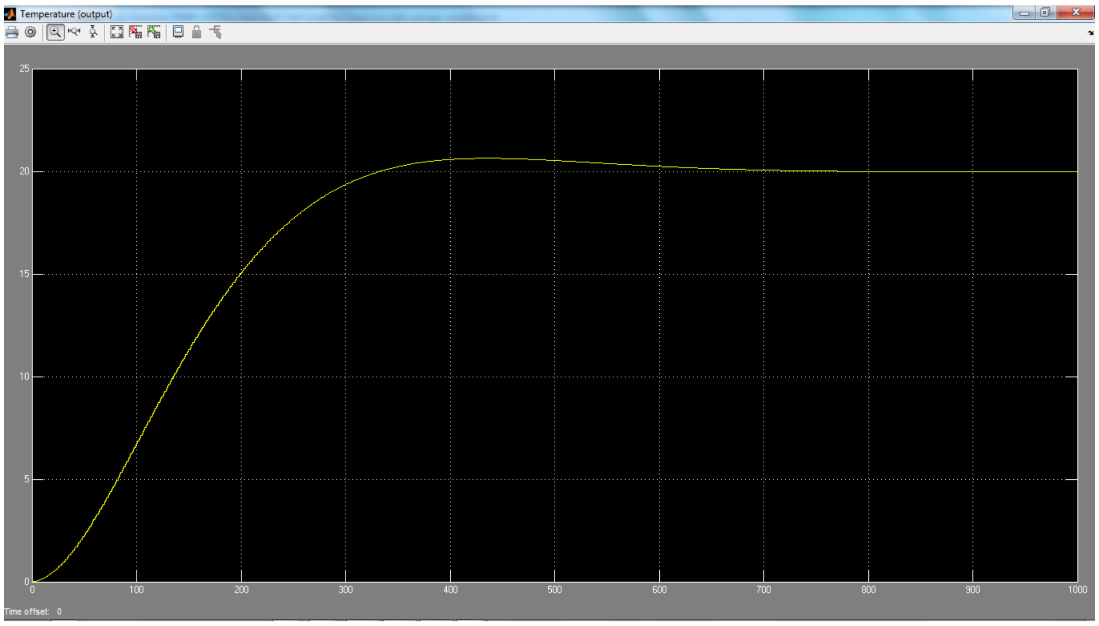


Figure 5.15: Functional performance of the thermostat model: Temperature

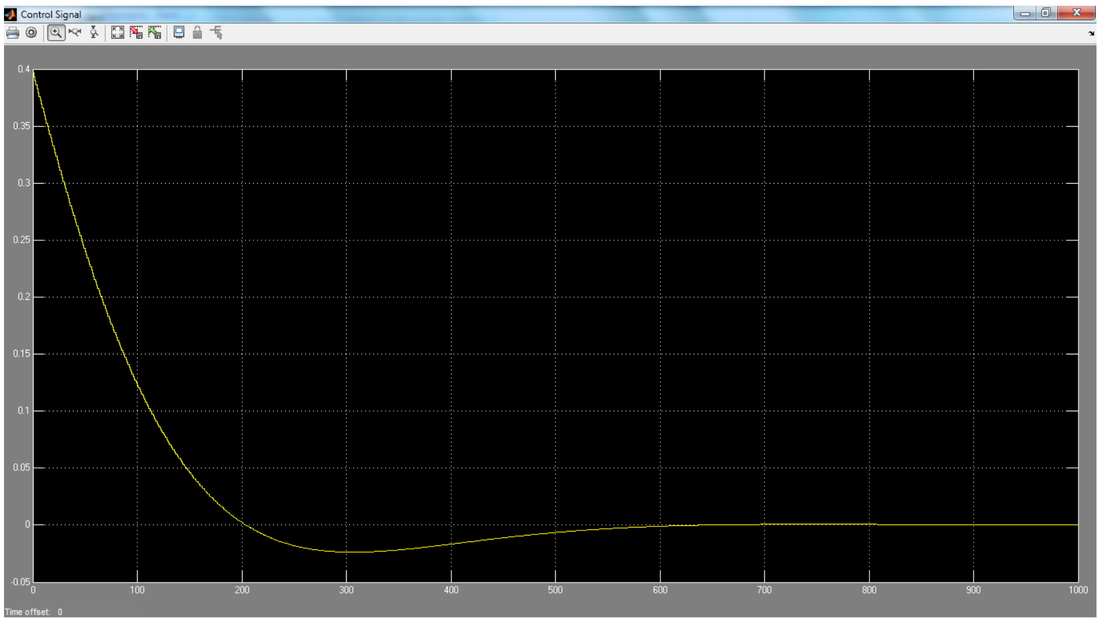


Figure 5.16: Functional performance of the thermostat model: Input signal u

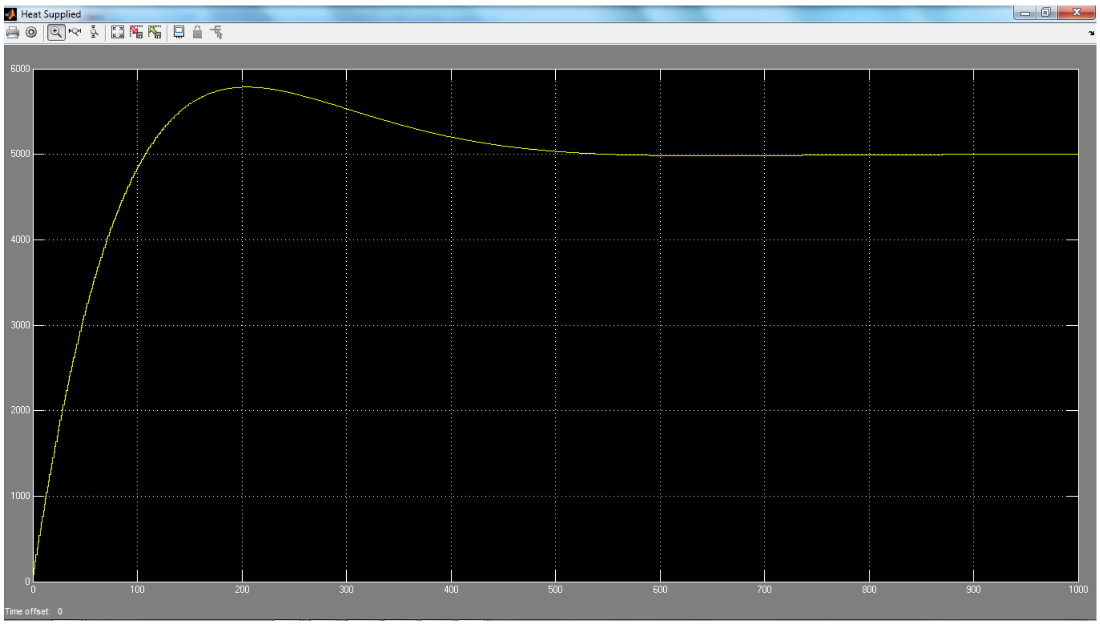


Figure 5.17: Functional performance of the thermostat model: Heat

Chapter 6

XSAP injection

In chapter 5 we presented our two use case examples, the simplified CAE and the room thermostat. Our line of exposition was to present the easier model first and then increase complexity. In this chapter we will swap the order around, presenting the injections of the thermostat example before the injections on the CAE. The reasons for that are manifold. First we will see that the injections on the thermostat are standard, once the structure is up. We can follow a procedural approach for that, it will highlight the importance of treating participants as stand-alone, livening our previous discussions up to a concrete instance. The injections on the CAE example are equally precious, but they will stimulate more questions and thus open our outlook to new research opportunities. The CAE has no physics going on, it is all about the topological transformations of the system and the temporal schedule of the participants, where interruption of services (like car unavailability) is the only thing that really counts. Once their description are identified, going from the thermostat style of dependability assessment to the CAE's is only a matter of elevating the description tier one step up.

For both examples we will present the fault modalities, the nominal nuXmv model, the fault injection and the results. All the experimental evaluations are performed on a Intel I5-2520M machine at 2.5GHz 3GB RAM, running the commands below.

```
python extend_model.py -v <model4injection>.smv <injection>.fei
python compute_ft.py -v --prop-name TLE_negated
--engine=[bdd, bmc, ic3, msat]
python view_ft.py -v
```

These will make more sense in a while. After that we will propose a novel approach for contract-based system design, that exploits duality of fault tree construction to construct what we will call *suggestion trees* or *planning tableaux*, based on whether they will be aimed at design or at planning. We will conclude by briefly expressing how we could combine the given approaches in the last segment of the chapter.

6.1 Thermostat injection

Given the static interconnections of our use case, we will be only interested in faults at the service level, abstracting away all other typical subtleties regarding service request[27], discovery and binding which are left as part for future work.

Generally, SOA service faults can either originate from internal malfunctions of the single participants or from connections dropping between communicating processes. In both cases we may assume that, once the binding between two participants is broken, it is re-established, somehow, some time later. This models the fact that an intermediate dynamic service broker interaction might come into the picture to provide another binding with either the same server or another providing the same functionalities, this bypassing our model's perspective unnoticed. From the single participant's perspective, this details regarding the drop of service are not relevant. This fits the SOA principle of loose-coupling and suggests that the error model can be approached using contract-based independence.

6.1.1 Fault Modalities

Faults in the Thermostat example or alike can happen in relation to the functional aspects of the system or to the delivered Quality-of-Service. Functionality issues can happen if one of the services that the participant is making use of is affected by behavioral issues. This case can be analyzed pretty much with standard techniques and requires no more than a conventional handling where participants are represented as embedded system components and communications as relations on ports. In this sense, system malfunction can be studied using standard model extension [23] or, faithful to the contractual view of the system, it can be treated as contract violation, using techniques similar to [20].

The story is different if we are willing to embrace Quality-of-Service requirements, i.e. non-functional properties, on our system. Those mainly relate to time properties of communication (such as latencies, jitter, round-trip time, ...) and are paramount for the correct serving of [distributed] Cyber-Physical Systems. In the case under analysis, delays in communications can stream out-of-date messages in the network, making the internal partial representation of the system of each participant inconsistent with reality, possibly leading the system itself to a crash, and with poor diagnostic information.

An effective way to inject model faults as latencies is thus essential for the system dependability assessment. We thought of two ways how this could be achieved, both important, in principle, for differentiating on how system design should be approached. The first option defines, for each participant, contracts including the latency view, using time in a global sense. For example the contract for the Thermocouple can be restated replacing the assumptions $[0 \leq heat \leq 70000]$ with the timed $[0 \leq heat \leq 70000]$ **and** $[t \leq 2.5]$, saying that the Thermocouple guarantees to provide a good functioning only if the time value is less than 2.5 (seconds), with the intended semantics of realizing the increase in temperature according to guarantees only if the heat signal reaches the Thermocouple early enough in the cyclic sequence of interaction.

This first option forces the participants' contract model to take on time-related variables, such as t, defined at the global scope. This is a hack, because it goes against the principle of loose-coupling and independence of contracts. Nonetheless it can be a valid option if the language used to specify the analysis constraints offers native support for time semantics on transitions and thus does not need the coerced introduction of a special variable intentionally for that purpose. Languages for the specification of timed dynamics of systems might be suitable for this kind of contract specification and the fault injection would operate changing transition times between system states at the global level. We are interested in a more localized understanding of the timing view.

Possibly, as our second option, we could create, for each input and output variable of a single participant, an associated time port, serving for the communication of latencies. In our example, the Thermocouple would have the additional input port t_{heat} (that would also appear as an output port for the Heater) and the output port t_T (input port for the Controller), bound to communicate the time that the message needs to reach the receiver from the moment it is created. We will refer to those additional ports as *latency ports*, even though the term latency is used in a broad sense to comprehend the more general notion of time.

Now, to be fair to our previous discussion, we would have to answer the question: Aren't latency ports themselves an hack to the model?. As a matter of fact there never is an explicit intentional communication of latency coming from the sender, being the sender unable to determine it in the first place. However, even from the perspective of a real-life implementation, once the receiver gets the message, it is in position to recover this information by using timing information contained in the message (timestamps) or by simply recording after how much time from the request, it gets the response from the addressed service[1]. Therefore we do not regard the action of giving value to a port as the action of the sender to stream a message on the channel (sender-side), but rather as the action of the receiver to acquire that information, regardless of where the information entered the channel from (receiver-side).

So latency ports are not an hack, they are a useful fancy way to look at the problem of modeling time information. They have the advantage of building a new part of the message grid, topologically isomorphic to the previous, with the nice property of leaving non-functional aspects in their own scope (i.e. without necessarily — although possibly — mixing with the functional specification of the model). We will make a step further from there though: we will introduce the network as a separate living entity, interposed to every communication, bound to scramble messages around.

To be more precise, the network deals with two sorts of messages, functional and

[1]Notice that this procedure relies on a global notion of time living in the implementation layer inside the architecture and does not affect how the model is to be regarded. So we assume that the globality of time is a notion embedded in the knowledge owned by each participant.

timing, that it treats differently. Functional messages are those flowing through the original ports of the model, while timing messages flow through latency ports. Every component controls as many latency ports as the number of its functional output channels. The value assigned to those ports — also called *internal latency ports* to indicate their values are not decided by network communications — is relative to the computation time needed for the component to produce the output, once the required inputs are available. In our running example, the Thermocouple needs 1 second between one sensing and the next, therefore it will put on its temperature internal latency port int_T the time needed to reach the 1 second bound from the time that already elapsed since the previous sensing. In other words the value on int_T will be given by $int_T = 1s - t_{heat}$.

The network is receptive of internal latency ports and decides by its own, following a schema independent from anything attached to the network, how to feed the single participant's latency ports. In our specific example, let S stand for Sensor, C for Controller and H for Heater. Supposing a communication latency of $0.01s$ between ant two participants and no internal latency for the controller and the heater (i.e. the computation time of the output is negligible), the network will pass on the controller's t_T port the value:

$$t_T = t_{C \to H} + t_{H \to S} + int_T + t_{S \to C} = 0.01s + 0.01s + 0.97s + 0.01s = 1.00s$$

Given that information, the controller only knows that, from its last sent message, the new input arrived 1.00 seconds after, and on that it will ground its computation. This machinery gives the network the prerogative to decide the communication times and leaves to components to decide of their own running times. So here, finally realized, is the separation between the functional system and the architecture.

The nominal version of the thermostat model is correct and it satisfies the top-level thermostat contract, as we have shown with our Matlab/SimuLink simulations of section 5.3.4. The latency ports only end up in being short negligible delays for the system, in nominal mode. Now, injections can happen on those ports as system delays. After injection, the thermocouple will use a new value for Δt in its computations, this time acquired from the network through the tainted latency ports. For example, if the *heat* message was so delayed to hit the thermocouple after 1 second, then the thermocouple would return a value sensed exactly after that time, and the whole system safety would be at stake, because the contract satisfactions of the single sub-components would no longer be guaranteed. We would like, eventually, to study the system under these sorts of degraded conditions[2].

In this new setting, the system specification can be completely made explicit at the interface level, hence presenting loose-coupling also for the non-functional part: by expressing assumptions and guarantees conditioned on properties over latency ports. Additionally, the ability of expressing non-functional properties with the same language as we specify functional properties, can lead to a cleaner handling of model extension by reusing, with a bit of care, the same faults present in the

[2]Notice that, virtually, we could put implementation controls on latency ports, to prevent system misbehaviour. We do not do that in order to keep the use case simple.

standard techniques, and therefore the same tools. This will allow us to model asynchronous communications in a synchronous communication tool, such as nuXmv.

The idea of introducing specialized additional ports for the specification of system components is not new. Already in [15] a similar idea is advised for the separation of concerns in multi-viewpoint contracts, where values of extra-ports — possibly related to communication timings — are randomly sampled from a probability distribution and injected by activation through a boolean variable. In contrast to our work they assume ports to be attached to components in a receptive way, left to an environment which itself cannot be controlled. Instead, our approach is to treat those ports like any other port and let them be either receptive or controlled, with no restrictions. In this sense, the network can really behave as the environmental mirror contract of the system, dispatching times for the communication latencies and realizing internal computational-time latencies of the single system participants. Notice that modeling the network as a individual system entity has particular relevance in the context of SOA, where service communications are as important as its participants.

6.1.2 Implementation

Now that we defined the fault modalities for our system we are ready to present the fault injection procedure. First we need to create the nominal model in nuXmv, then provide latency ports for each components and finally inject the faults related to timings. We will go through those points methodically, one by one. We will not present any result of the fault injection to the functional view, because it is beyond the scope of our treatment. We already presented in chapter 1 some of the existing works using model extension for the dependability analysis, [54] contains one using XSAP and more are to come. In [20] Bozzano and al. present an approach to functional dependability based on contracts, we discussed in section 1.3 that extensions of their approach with more complex fault description is part of future research.

In this chapter we will limit ourselves to properties more distinctive of service distribution. This will not cause loss in generality on our approach because the injection procedure that we shall soon present over latency ports are equally applicable to functional ports, as already mentioned at the end of the previous section, being their distinction only conventional.

Nominal SMV model

In this section we present the nominal model of the thermostat, inspired by our functional Matlab/SimuLink prototype that we mentioned in section 5.3.4.

The specification has three modules, corresponding to the thermocouple, the heat controller and the heat actuator, linked together by the relations in the main module. `DEFINE`s represent definitional pieces of code to substitute to others, similar to macros in most common programming languages. They are used to express system parameters and compact parts of formulae.

```
MODULE ThermoCouple(heat)

DEFINE
   alphacond := 5.0;   -- W/(m^2*K)
   surface := 50.0;    -- m^2
   Ta := 0;            -- Celsius degrees
DEFINE
   Cs := 1000.0;    -- J/(Kg*K)
   m  := 35.0;      -- Kg
   deltaT := 1.0;   -- s

VAR
   Tr : Real; -- Temperature of the room (Celsius degrees)
   time : Real;

DEFINE
   der_dissipation := alphacond*surface*(Tr - Ta);

   init(Tr) := 0.0;
   next(Tr) := ((deltaT*heat - der_dissipation*deltaT)/(Cs*m)) + Tr;

   init(time) := 0.0;
   next(time) := time + deltaT;
```

```
MODULE Controller(Tr)
VAR
   u : Real; -- [-1, 1]

DEFINE
-- K parameters of PID are defined based on SimuLink tuning --
   Kp := 0.02;
   Ki := 0.0;
   Kd := 1.5;

DEFINE
   Tref := 20;
   err := (Tref - Tr);
   diff_err := (err - last_err);

VAR
   sum_err : Real;
   last_err : Real;

ASSIGN
   init(u) := 0;
   next(u) := Kp*err + Ki*sum_err + Kd*diff_err;

   init(sum_err) := 0;
   next(sum_err) := sum_err + err;
```

```

  init(last_err) := Tref;
  next(last_err) := err;
```

```
MODULE Heater(u)
VAR
  heat : Real; -- [0, 10000]

DEFINE
  max_heat := 10000; -- maximum heat supply
  der_heat := heat+200*u; -- how the heater *actuates* the signal

ASSIGN
  init(heat) := 0; -- start from no heat supply
  next(heat) := (der_heat < max_heat) ? der_heat : max_heat; -- bounded
```

```
MODULE main
VAR
  t : ThermoCouple(h.heat);
  c : Controller(t.Tr);
  h : Heater(c.u);
```

This model is compliant to the Matlab/SimuLink system that we sketched in section 5.3.4, that will be our baseline[3]. We will have to make two main modifications, first to introduce our fault modalities (through latency ports) and second to make it usable by XSAP, which is not able to deal with infinite precision models. We will need some way to discretize the model and realize it in finiteness.

Attaching latency ports

The functional model shall be enriched with latency ports and internal latency ports for interfacing with the network. To do this the thermostat model has to be enriched with the following features:

1) Latency ports as described above, one for each input and output port of the system components, including internal latency ports and component's internal computation times (e.g. 1.0 second for the Thermocouple)
2) A network module, a new entity to which every component is attached. It forwards values on functional ports directly to components and decides values for the input latency ports, based on the values of internal latency ports. Ideally, the network would be in charge to simulate the real dynamics of the network with respect to the timings view. In our example the network puts on latency ports the time for a message to round-trip to the component once it leaves, but other dynamics are not to be excluded.

[3]This correspondence between the two systems can be made explicit by simulating our latest model in nuXmv and comparing the two results for, e.g., the temperature. We do not report this comparison here to keep the presentation lighter.

3) Introduction of sensitiveness to time latencies on variable `deltaT` in the ThermoCouple module. The idea is to continue heating with the previous heat until the next heat message arrives (for `deltaT_before_arrival` secs), then start with the new heat supply (for `deltaT_after_arrival` secs).

We assume that the module interaction starts from the ThermoCouple sending *immediately* the sensed temperature on the network. It follows that the system, making a message round-trip for each time step, synchronizes on the ThermoCouple module. As a result the perceived time is delayed from one component to the other. This will not affect our intended model because the only use of time that we make is confined in the Thermocouple. This is a featured aspect of the methodology that has to be taken into account if more complex dynamics are supplied to systems: perceived time is intentionally different from one component to the other.

Latency ports should never get negative values, but neither 0 is acceptable, because a minimal communication delay should always be put into modeling. For the interactions of our use case, as in the previous section, we can set a fictitious transmission time of 0.01 seconds, which comprises the round-trip time of the SOAP service invocation and retrieval. This will be the default value to assign to latency ports and it is arbitrary to our network model.

One latency port is exceptional. As we mentioned in the use case specification, the Thermocouple senses the environment at the rate of one second. This information needs to be modeled as the Controller receiving the temperature message not earlier than every second. Accounting for the round-trip time of request and assuming that the sensing delay time of one second is already started by when the request reaches the Thermocouple, it takes exactly one second for the service message to reach the Control Device. This can be modeled by setting the value of the Thermocouple's internal latency port to $int_T = 1.0$ (we already saw this briefly in the previous section).

The enriched model specification follows, with the new parts highlighted in red:

```
MODULE ThermoCouple(heat, LP_heat)   -- LP = "Latency Port"

DEFINE
   alphacond := 5.0;    -- W/(m^2*K)
   surface := 50.0;     -- m^2
   Ta := 0.0;           -- K
DEFINE
   Cs := 1000.0;     -- J/(Kg*K)
   m  := 35.0;       -- Kg
   deltaT := deltaT_before_arrival + deltaT_after_arrival;
   deltaT_before_arrival := LP_heat;
   deltaT_after_arrival := ILP;

VAR
   Tr : Real;

DEFINE
```

```
  der_dissipation := alphacond*surface*(Tr - Ta);

DEFINE
  computation_time := 0.0;
  ILP := ((computation_time >= LP_u) ? computation_time - LP_u : 0.0);


ASSIGN
  init(Tr) := 0.0;
  next(Tr) := ((deltaT*heat - der_dissipation*deltaT)/(Cs*m)) + Tr;
```

```
MODULE Controller(Tr, LP_Tr) -- LP : "Latency Port"

VAR
  u : Real; -- [-1, 1]

DEFINE
  Kp := 0.02;
  Ki := 0.0;
  Kd := 1.5;

DEFINE
  Tref := 20;
  err := (Tref - Tr);
  -- sum_err defined as a variable
  diff_err := (err - last_err);

VAR
  sum_err : Real;
  last_err : Real;

DEFINE
  computation_time := 0.0;
  ILP := ((computation_time >= LP_u) ? computation_time - LP_u : 0.0);

ASSIGN
  init(u) := 0;
  next(u) := Kp*err + Ki*sum_err + Kd*diff_err;

  init(sum_err) := 0;
  next(sum_err) := sum_err + err;

  init(last_err) := Tref;
  next(last_err) := err;
```

```
MODULE Heater(u, LP_u)  -- LP = "Latency Port"

VAR
  heat : Real;
```

```

DEFINE
  max_heat := 10000;
  der_heat := heat+200*u;

DEFINE
  computation_time := 0.0;
  ILP := ((computation_time >= LP_u) ? computation_time - LP_u : 0.0);

ASSIGN
  init(heat) := 0; -- start from no heat supply
  next(heat) := (der_heat < max_heat) ? der_heat : max_heat;
```

```
MODULE Network(ILP_Heater, heat, ILP_Thermocouple, Tr, ILP_Control, u)
DEFINE -- nominal latencies
  latency_heat := 0.01;
  latency_Tr   := 0.01;
  latency_u    := 0.01;

VAR -- Latency Ports
  LP_heat : Real;
  LP_Tr   : Real;
  LP_u    : Real;

DEFINE _Tr_ := Tr; _u_ := u; _heat_ := heat; -- technicality

ASSIGN
  init(LP_heat) := 0.0;
  init(LP_Tr)   := latency_Tr;
  init(LP_u)    := latency_Tr + ILP_Control + latency_u;

  next(LP_heat) := latency_Tr + ILP_Control + latency_u
                            + ILP_Heater + latency_heat;
  next(LP_Tr)   := latency_u + ILP_Heater + latency_heat
                           + ILP_Thermocouple + latency_Tr;
  next(LP_u)    := latency_heat + ILP_Thermocouple + latency_Tr
                              + ILP_Control + latency_u;
```

```
MODULE main
VAR
  t : ThermoCouple(net._heat_, net.LP_heat);
  c : Controller(net._Tr_, net.LP_Tr);
  h : Heater(net._u_, net.LP_u);

  net : Network(h.ILP, h.heat, t.ILP, t.Tr, c.ILP, c.u);
```

Discretization

As formerly mentioned, before feeding our model to XSAP we need to finitize its structure. This is possible for two reasons: boundedness (the model is supposed to be run for a finite number of steps — 600 time units in our case) and coarseness (our model does not require high precision arithmetics, this because it is based on strong simplifications in its very definition already).

We can proceed as follows:

1) transform *Real* variables in finite precision variables bounded by fixed point rational numbers

2) multiply fixed point numbers by a common number (power of 10) to make them all valued in some bounded integer range.

In order to streamline the state space of model we also removed the integral gain because it was pointless from an implementation point of view to have a dedicated state variable for a gain with coefficient $K_i = 0$: its contribution to the control signal was in fact all the time null:

```
next(u) := Kp*err + Kd*diff_err;
```

The following snippet shows our attempt where we kept two significant digits for the temperature value and times while applying the two points above. The @ symbol indicates 10^-2, therefore $1s = 100@s$, $1K = 100@°\text{C}$. For sake of convenience we only show the changes in the module for the Thermocouple; for the other modules, changes are alike.

```
MODULE ThermoCouple(heat, LP_heat) -- heat in W; LP_heat in @s
DEFINE
  alphacond := 5;    -- W/(m^2*K)
  surface := 50;     -- m^2
  Ta := 0;              -- Celsius degrees
DEFINE
  Cs := 1000;     -- J/(Kg*K)
  m  := 35;       -- Kg
  deltaT := deltaT_before_arrival + deltaT_after_arrival; -- @s
  deltaT_before_arrival := LP_heat; -- @s
  deltaT_after_arrival := ILP; -- @s

VAR
  Tr : 0..2500; -- Temperature of the room

DEFINE
  computation_time := 100; -- @s
  der_dissipation := (alphacond*surface*(Tr - Ta))/100; -- W (@J/@s)

DEFINE ILP := 100;

ASSIGN
  init(Tr) := 0;  -- @Celsius
  next(Tr) := (((deltaT*heat - deltaT*der_dissipation))/(Cs*m)) + Tr;
```

This procedure rises more than one problem when brought to reality: first and foremost, the state space explosion. Representing the whole system with *Real*s let nuXmv exploit its SMT reasoning features; this reduced the problem to a SMT version of a simulation problem, which nuXmv could handle easily. Bounding model values with integer numbers calls for the state space to abandon space connectedness, thus obliging the system to work in a grid of separated values. Symbolic reasoning can still help but it might not be enough.

Other problems relate performance and circuitry, because when working with bounded integers, those become huge. Despite our model is inherently bound to simulation, the model checker is not aware of it and starts constructing a state space for the system as it were time-unbounded. This problem comes along with state explosion. Moreover integer division and multiplication have huge counterparts in the logic of bits, hence the problem with circuitry.

The solution to the problems sketched in the last paragraphs lays in the adoption of the word nuXmv type, that we present in the next section.

From bounded integers to words

The model that we presented with bounded integers explains rather intuitively how the conversion to the discrete domain can be achieved. However, the presented model cannot even be simulated using bounded resources, because the variable domains and operators circuitry are too complex to model check. The following part shows how we faced the problem, namely by translating the model further to the word domain.

Words, or bitvectors as they are sometimes called, are similar to bounded integers, but they are not strictly bound. To be more precise, they are formal binary representations of integer numbers that are allowed to overflow and underflow, as in digital systems. They come provided with standard operators (addition, subtraction, multiplication, ...) but their semantics is different than that of bounded integers[4]. They also come with additional specific operators, such as left-shift (`<<`) and right-shift (`>>`). These are respectively semantically equivalent to multiplying and dividing by powers of 2, but are notoriously faster and more natural for the binary world.

The facility of using word types does not come with no pain and struggle, because everything must be hand-crafted very accurately to avoid bit overflows. To implement the final version of our model, that is presented in the following, we changed integer domains to word domains and multiplications/divisions by constants to fixed bit-shifting operations. To show our methodology we also set delay times on latency ports as powers of two and expressed them as numbers in [0..10]. This made us lose some precision, although not enough to invalidate our model.

For sake of completeness we report here the module for the Thermocouple, which

[4]A typical example on the theory of bitvectors is the property $\neg\forall w_1, w_2 \,.\, w_1 + w_2 \geq w_1$ that can be proved by showing $1 + 1 = 0$ on words of length one.

presents all interesting features hard-coded. It will not be familiar to an inexperienced eye but we tried to keep it as much understandable as possible.

```
MODULE ThermoCouple(heat, LP_heat) -- LP in logarithmic mode
DEFINE
   SHIFT_DISS := 8; -- *alphacond*surface = *250 ~= *2^8 = <<8;
   SHIFT_TR := 15; -- /(Cs*m) = / 35000 ~= / 2^15 = >>15;

DEFINE Twsize := 13;
VAR Tr : unsigned word[Twsize];

DEFINE
   der_dissipation := (extend(Tr,SHIFT_DISS)<<SHIFT_DISS);

--- Timings! -------------------------------
DEFINE
   computation_time := 7;
   SHIFT_deltaT := SHIFT_deltaT_before_arrival
                 + SHIFT_deltaT_after_arrival;
   SHIFT_deltaT_before_arrival := LP_heat;
   SHIFT_deltaT_after_arrival := ILP;
   ILP := ((computation_time >= LP_heat)
           ? computation_time - LP_heat
           : 0
          );

VAR time : unsigned word[16]; -- assume simulation time <= 2^16-1
ASSIGN
   init(time) := uwconst(0, 16);
   next(time) := (time>=uwconst(60000, 16))
                  ? time
                  : time + (uwconst(1, 16)<<SHIFT_deltaT);
--------------------------------------------


DEFINE Tr_prime :=
    unsigned(resize(
              ((extend(signed(
                          extend(heat,
                                 SHIFT_DISS - (14 - Twsize)))
                 , 1)
             - extend(signed(der_dissipation>>7), 1)
              )<<SHIFT_deltaT)>>SHIFT_TR, Twsize)
             + signed(Tr)
            );
ASSIGN
   init(Tr) := uwconst(0, Twsize);
   next(Tr) := Tr_prime;
```

After the transformation we were able to run and simulate the model using standard bmc; it took it only a few instants to finish the run using a bound of 600.

Model extension and Fault Tree construction

In section 6.1 we proposed for the fault modalities modifications of the values on latency ports. In this section we will perform this operation using XSAP extension and fault tree construction. In the case at study we will assume that the faults are transient and consist in a gradual increase in the network times of communication between participants.

It is our deliberate choice to treat each communication separately. In a conventional setting the network would experience problems, such as congestion, in a shared manner and it is often responsibility of the network protocol designers to balance the resulting load with fairness. In our abstraction, however, the network is intended in a broader sense, spread across different networks that experience their own issues way before and more intensively than they experience issues as a group. Virtually, each communication is given by a sequence of action that require the service to a broker and need to bind before execution. Even when the connection is made permanent, at any moment the connection might be lost and the research of a new service provider might hinder response times on that individual connection. Although we could have provided the network agent with synchronous delays, we thereby preferred not to, for the sake of generality of our approach.

The first step is to enrich the nominal model with the hook variables. There will be two for each latency port: one for tracking the fault in degraded mode and one for the recovery when full power of the connection is re-established. Importantly, hooks for the timing view are all put on the network component and the functional view remains intact as it were.

```
MODULE Network(ILP_Heater, heat, ILP_Thermocouple, Tr, ILP_Control, u)
DEFINE -- nominal latencies (logarithmic mode)
  latency_heat := 1;
  latency_Tr   := 1;
  latency_u    := 1;

VAR -- Latency Ports in log mode
  LP_heat : 0..10;
  LP_Tr   : 1..10;
  LP_u    : 1..10;

--- Injection variables ----------------
IVAR degraded_heat : boolean;
IVAR degraded_Tr : boolean;
IVAR degraded_u : boolean;

IVAR nominal_heat : boolean;
IVAR nominal_Tr : boolean;
IVAR nominal_u : boolean;
```

```
------------------------------------------------------------

ASSIGN
  init(LP_heat)  :=  0;
  init(LP_Tr)    :=  latency_Tr;
  init(LP_u)     :=  latency_Tr + ILP_Control + latency_u;

  next(LP_heat)  :=  latency_Tr + ILP_Control + latency_u
                            + ILP_Heater + latency_heat;
  next(LP_Tr)    :=  latency_u + ILP_Heater + latency_heat
                            + ILP_Thermocouple + latency_Tr;
  next(LP_u)     :=  latency_heat + ILP_Thermocouple + latency_Tr
                               + ILP_Control + latency_u;

DEFINE _Tr_ := Tr; _u_ := u; _heat_ := heat;
```

In order to specify the Top-Level property around which the dependability analysis will be established, we devised a simple monitor whose task is to supervise the system and, int this particular case, trigger a timeout event if 600 seconds have elapsed without the reference temperature of the room being reached.

```
MODULE Monitor(Tr, time)
  DEFINE MAX_TIME := uwconst(60000, 16);
  DEFINE timeout := (time>=MAX_TIME) & (Tr<uwconst(2000, 13));
```

```
MODULE main
VAR
  t : ThermoCouple(net._heat_, net.LP_heat);
  c : Controller(net._Tr_, net.LP_Tr);
  h : Heater(net._u_, net.LP_u);

  net : Network(h.ILP, h.heat, t.ILP, t.Tr, c.ILP, c.u);
  monitor : Monitor(t.Tr, t.time);
```

At this point we can ask XSAP to inject the model with faults and construct the fault tree for the property *timeout*:

```
INVARSPEC NAME TLE_negated := !monitor.timeout;
```

The extension file is specified with three fault modes per latency port: stuck-at-value 2, 5, 10. Notice that they are in logarithmic mode and after that bit shifting they correspond respectively to 2^2, 2^5 and 2^{10} seconds.

We only report the injection for one port, for the other two it goes the same:

```
FAULT EXTENSION FE_THERMOSTAT
  -- Latency port can temporary work in degraded conditions (delay in [2..10]) --
  EXTENSION OF MODULE Network
     SLICE __heat__ AFFECTS LP_heat WITH
```

```
        MODE delay2 : Transient StuckAtByValue_I (
                 data      term << 2,
                 data      input << LP_heat,
                 data      varout >> LP_heat,
                template   self_fix = fixed,
                 event     failure >> degraded_heat,
                 event     fixed >> nominal_heat
                );
        MODE delay5 : Transient StuckAtByValue_I (
                 data      term << 5,
                 data      input << LP_heat,
                 data      varout >> LP_heat,
                template   self_fix = fixed,
                 event     failure >> degraded_heat,
                 event     fixed >> nominal_heat
                );
        MODE delay10 : Transient StuckAtByValue_I (
                 data      term << 10,
                 data      input << LP_heat,
                 data      varout >> LP_heat,
                template   self_fix = fixed,
                 event     failure >> degraded_heat,
                 event     fixed >> nominal_heat
                );
```

We run the analysis, it never terminates, no matter the engine. To give an intuition on how much time the simulation needs on our machine, using bmc it took 9 minutes to simulate on the first 100 seconds against reference temperature 1°C, while time increased to 16 minutes to simulate the first 110 seconds. Unfortunately, by introducing faults in the model, XSAP literally blows up the state space every time a step is advanced, because every communication is allowed to undergo a latency delay.

We acknowledged some improvements if the reference temperature was increased with the bound because short minimal traces forming the cut sets could be found earlier by the optimized versions of the tool, but certainly never it stood out as much as needed. We could easily estimate and persuade ourselves that we would never be able to reach the bound of 600 seconds any time soon.

So we ask, is that it?

Contracts and scale

The fault tree that we obtained running 100 timesteps against $Tr = 1°C$ is shown in figure 6.1. What the fault tree tells us is that in order to fail the given property (not being able to reach 1°C in 100 seconds) one has to put very high delays on the heat latency port, at least in the order of 2^{10} centiseconds according to our available fault modes. Interestingly enough, every fault tree that we obtained by running the analysis with different parameters in the monitor, including simulation time and

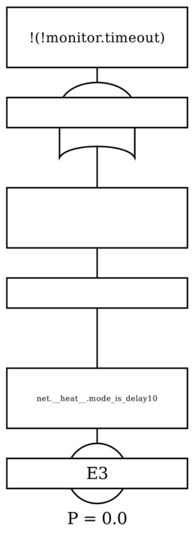


Figure 6.1: The Fault Tree constructed by XSAP for the Thermostat model.

reference temperature, returned the same fault tree. The reason for this lays in of the network structure: every communication to the sensor has to go through the heat latency port, therefore there the faults concentrate. This fact could be noticed even more easily by looking at the interface specification of the component: at its contract. Here our next idea.

The complexity of the dependability analysis can be amazingly lowered by focusing at the single components' facet. By our definition of contracts, in saturated form, if each module implements its own, then the whole system satisfies their composition. Although, if one does not satisfy the contract, then the system is not compelled to follow the overall contract's guarantees. Therefore, studying the fault modalities of the single contracts allows to study the fault modalities of the system as a whole.

We developed a model for the Thermocouple, the only component specifying timing restrictions on its contract, with the precise intention of studying its timing view with respect to its contract. We do not need the specification of the heater or the controller, because here only the Thermocouple has the focus. The point now will not be to simulate the entire system evolution, but rather that *every possible implementation deriving from the Thermocouple satisfies the contract of the Thermocouple.* Refinement checking of section 5.3.3 did the rest for us.

In order to check contract satisfaction for every derived component we have to check that all allowed values on the input ports and state variables — the *heat* port and Tr variable in our case — lead to contract in any case to contract satisfaction. We need the network because it has the hooks for the fault injection.

```
MODULE ThermoCouple(heat, LP_heat)
DEFINE
```

```
  SHIFT_DISS := 8;
  SHIFT_TR := 15;

DEFINE
  der_dissipation := (extend(Tr,SHIFT_DISS)<<SHIFT_DISS);

--- Timings! ------------------------------
DEFINE
  computation_time := 7;
  SHIFT_deltaT := SHIFT_deltaT_before_arrival
                + SHIFT_deltaT_after_arrival;
  SHIFT_deltaT_before_arrival := LP_heat;
  SHIFT_deltaT_after_arrival := ILP;
  ILP := ((computation_time >= LP_heat)
          ? computation_time
          - LP_heat : 0
         );
VAR time : unsigned word[16];
ASSIGN
  init(time) := uwconst(0, 16);
  next(time) := (time>=uwconst(60000,16))
                ? time
                : time + (uwconst(1, 16)<<SHIFT_deltaT);
------------------------------------------

-- With random heat and random (+bounded) init(Tr) this tracks all
-- all possible evolutions of Tr; notice we are interested in one
-- step of evolution -- only (input->computation->output)
DEFINE Twsize := 13;
VAR Tr : unsigned word[Twsize];
ASSIGN next(Tr) := Tr_prime;
DEFINE Tr_prime :=
   unsigned(resize(
              ((extend(signed(
                        extend(heat,
                               SHIFT_DISS - (14 - Twsize)))
                 , 1)
              - extend(signed(der_dissipation>>7), 1)
              )<<SHIFT_deltaT)>>SHIFT_TR, Twsize)
              + signed(Tr)
             );
INIT Tr <= uwconst(3000, Twsize);
FROZENVAR Tr_initial : unsigned word[Twsize]; INIT Tr_initial = Tr;
```

```
MODULE Network()
--- Injection variables ----------------
IVAR degraded_heat : boolean;
IVAR nominal_heat : boolean;
------------------------------------------
```

```
VAR LP_heat : 0..10;
ASSIGN LP_heat := 0;

DEFINE heatwsize := 14;
VAR _heat_ : unsigned word[heatwsize];
INVAR _heat_ <= uwconst(10000, heatwsize);
```

```
MODULE Monitor(heat)
VAR bigheat : boolean;
ASSIGN
  init(bigheat) := heat>uwconst(7500+2100, 14);
  next(bigheat) := bigheat & heat>uwconst(7500+2100, 14);

```

```
MODULE main
VAR
  t : ThermoCouple(net._heat_, net.LP_heat);
  net : Network();
  monitor : Monitor(net._heat_);

INVARSPEC NAME TLE_negated :=
        ((t.time>=uwconst(1000,16)) & (monitor.bigheat))
                    -> (t.Tr>t.Tr_initial+uwconst(60, 13));
-- We did not specify the other TLE_negated of the contract
-- because we had double precision only for Tr.
```

Now the analysis takes no time to perform and still it returns the same fault tree of figure 6.1, improving the system dependability assessment astoundingly[5].

As a final step we enriched the `.fei` file with all timing injections from 2 to 10, that now could be tackled by the systems in really a few instants. This way we were able to obtain the Fault Tree of figure 6.2, that allows to determine that the system is safe as long as the heat message reaches the heat port of the Thermocouple in less than or equal to 2^7 centiseconds since the last departure of T_r. This is perfectly in line with the model: up to a delay of 2^7 log-centiseconds — which is our approximation of 100 in logarithmic scale — the system behaves as if no delay was there, i.e. it is resilient. However, as delays go from 2^7 to 2^8, so that the system experiences a delay of 2^7 beyond the Thermocouple computation time, the system starts to malfunction. Our interpretation suggests that the system can be read very easily and its resiliency to faults determined. This is yet another evidence that the approach that we used is valuable to our aims.

Notice that this time we did not assess dependability on the overall system, directly. Rather we focused on the possible inputs and controlled outputs of the single

[5]From an accurate reading of the invariant property specification, the careful reader might notice that we did not *really* verify the contract of the Thermocouple, but a necessary condition deriving from it (or a satisfaction on average if the reader prefers). This stacks on top of all other approximations and simplifications that we did. Once again, our study is about the methodology rather than on perfect accuracy of the models.

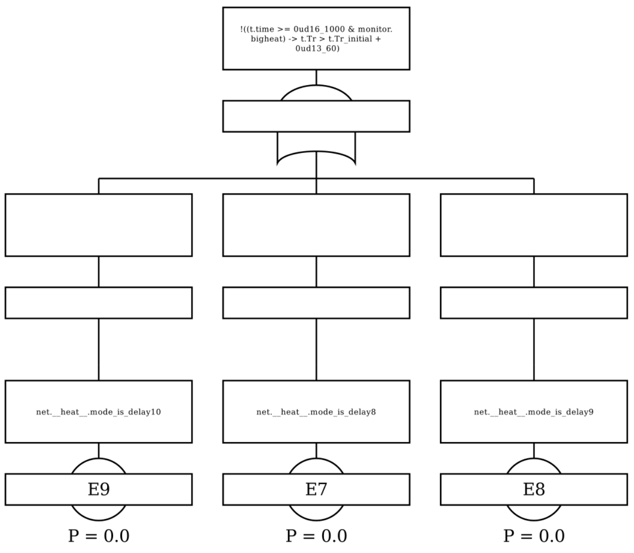

!((t.time >= 0ud16_1000 & monitor.bigheat) -> t.Tr > t.Tr_initial + 0ud13_60)
net.__heat__.mode_is_delay10
net.__heat__.mode_is_delay8
net.__heat__.mode_is_delay9
E9
E7
E8
P = 0.0
P = 0.0
P = 0.0

Figure 6.2: The Fault Tree constructed by XSAP for the Thermostat model: enriched version exploiting contracts.

component, possibly including behaviours that will never happen in the combined system (e.g. arbitrary heat supplies). We stress that this is not a shortcoming but an empowerment, because the resulting fault tree can now enjoy all the good qualities resulting from dealing with contracts, including reusability, possible hierarchical constructions and separation of responsibilities. It is the very point of contracts to not track exact behaviours of a compound system but being able to interact with any, now this same power is exploitable by our analysis of system dependability.

6.2 CAE injection

The CAE example is very different from the thermostat example and thereby it will need other fault modalities to get its dependability assessed. As we stated in section 5.2, the focus of the CAE with respect to our concerns is on the service interactions and their evolutionary dynamics.

As opposed to the thermostat, which assumed everything was fixed in the topology, here there will be parts of the system, either connections or components, whose presence is not to be taken for granted. However, when existing participants will be given a task on existing connections, like going to the district place on fire or extinguishing the fire once there, that task will be accomplished no matter what happens to them internally. It is unthinkable, in fact, to model the internal structure of a car going to the target place, with all the roads of the physical infrastructure, with the human control and all that.

6.2.1 Fault Modalities

The fault modalities that this system will be subject to will be sudden, not necessarily permanent, disappearances of services (boolean unavailability).

6.2.2 Implementation

As for the thermostat example we will first create our nominal model, inject faults and see the results. The CAE system is composed of the three agent modules for the District, the Fire Station and 5 Fire Fighting Cars, linked together in the main according to the specifications of section 5.2. Unlike the case of the thermostat, here we will not need a specific network agent for the interactions because we will not need to attach non-functional behaviours such as timing. The introduction of the network or a similar conception is anyhow possible and would have the advantage of having all fault modalities gathered together in one entity.

Nominal SMV model

The nominal SMV model of the CAE represents the system and its interconnections, assuming no faults and that everything goes smoothly according to plans.

There are two definition at the beginning of the District specification namely `MAX_FIRES` and `MAX_COMING_NEWFIRES`. These represent respectively the maximum number of fires of the system and the maximum number of fires that can materialize in the district. The number of fires at each time step is computed as the sum of the fires at the previous step excluding those that have been extinguished and the new detected fires.

The policy implemented by the Fire Station is to send one car for each acknowledged fire. In particular the car car_i is sent on district if the number of fires of the district is greater than the number of cars already sent on district and car_i is the first available car in the interval $[1, i]$. Once cars are sent they leave the Fire Station; once they reach the fire they extinguish it in the next step. After that they acknowledge the Fire Station and the District as described in section 5.2. Acknowledgments from the single Fire Fighting Cars to the District are sent cumulatively using the `count` function, which decides how many signals are set to the value `TRUE`.

```
MODULE District(extinguished_fires)
DEFINE
   MAX_FIRES := 5;
   MAX_COMING_NEWFIRES := 2;

VAR new_fires : 0..MAX_COMING_NEWFIRES;
VAR
   fires : -1..MAX_FIRES;          -- value -1 is unreachable (technicality)
   newfire_counter : 0..(MAX_FIRES+1); -- (MAX_FIRES+1) = no more fires

DEFINE
    new_coming_fires :=
            case
                next(newfire_counter<=MAX_FIRES) : new_fires;
                --- previous guard is passed ---
                newfire_counter<=MAX_FIRES : MAX_FIRES - newfire_counter;
                TRUE : 0;
            esac;

DEFINE fires_prime := fires - extinguished_fires + new_coming_fires;
ASSIGN
   init(fires) := 0;
   next(fires) :=
        case
          (0 <= fires_prime) & (fires_prime <= MAX_FIRES) : fires_prime;
          TRUE: -1; -- unreachable!
        esac;
   init(newfire_counter) := 0;
   next(newfire_counter) := ((newfire_counter+new_fires <= MAX_FIRES) ?
                              newfire_counter+new_fires :
                              MAX_FIRES+1
                             );

LTLSPEC G (fires != -1); -- check unreachability
```

```
MODULE FireStation(fires, car1_coming_back, car2_coming_back,
                    car3_coming_back, car4_coming_back,car5_coming_back)

VAR
   car1_isHere : boolean;  car1_go : boolean;  car1_gone : boolean;
   car2_isHere : boolean;  car2_go : boolean;  car2_gone : boolean;
   car3_isHere : boolean;  car3_go : boolean;  car3_gone : boolean;
   car4_isHere : boolean;  car4_go : boolean;  car4_gone : boolean;
   car5_isHere : boolean;  car5_go : boolean;  car5_gone : boolean;

DEFINE __going_cars__ := count(
                                 car1_go | car1_gone,
                                 car2_go | car2_gone,
                                 car3_go | car3_gone,
                                 car4_go | car4_gone,
                                 car5_go | car5_gone
                               );

ASSIGN
   init(car1_isHere) := TRUE;        init(car1_go) := FALSE;
   init(car2_isHere) := TRUE;        init(car2_go) := FALSE;
   init(car3_isHere) := TRUE;        init(car3_go) := FALSE;
   init(car4_isHere) := TRUE;        init(car4_go) := FALSE;
   init(car5_isHere) := TRUE;        init(car5_go) := FALSE;

   next(car1_isHere) := car1_coming_back | (car1_isHere & !car1_go);
```

```
  next(car2_isHere) := car2_coming_back | (car2_isHere & !car2_go);
  next(car3_isHere) := car3_coming_back | (car3_isHere & !car3_go);
  next(car4_isHere) := car4_coming_back | (car4_isHere & !car4_go);
  next(car5_isHere) := car5_coming_back | (car5_isHere & !car5_go);

  next(car1_go) := (fires - __going_cars__ > 0) & next(car1_isHere);
  next(car2_go) := (fires - __going_cars__
                         - count(next(car1_isHere)) > 0
                    ) & next(car2_isHere);
  next(car3_go) := (fires - __going_cars__
                         - count(next(car1_isHere),
                                 next(car2_isHere)) > 0
                    ) & next(car3_isHere);
  next(car4_go) := (fires - __going_cars__
                         - count(next(car1_isHere),
                                 next(car2_isHere),
                                 next(car3_isHere)) > 0

                    ) & next(car4_isHere);
  next(car5_go) := (fires - __going_cars__
                         - count(next(car1_isHere),
                                 next(car2_isHere),
                                 next(car3_isHere),
                                 next(car4_isHere)) > 0
                   ) & next(car5_isHere);

  next(car1_gone) := car1_go & next(!car1_coming_back);
  next(car2_gone) := car2_go & next(!car2_coming_back);
  next(car3_gone) := car3_go & next(!car3_coming_back);
  next(car4_gone) := car4_go & next(!car4_coming_back);
  next(car5_gone) := car5_go & next(!car5_coming_back);
```

```
MODULE FireFightingCar(sent_now)
VAR
  just_arrived : boolean;
  fire_extinguished_ack : boolean;
  going_back_ack : boolean;

ASSIGN
  init(just_arrived) := FALSE;
  next(just_arrived) := (sent_now=TRUE);

  init(fire_extinguished_ack) := FALSE;
  next(fire_extinguished_ack) := just_arrived & !fire_extinguished_ack;

  init(going_back_ack) := FALSE;
  next(going_back_ack) := fire_extinguished_ack & !going_back_ack;
```

```
MODULE main
```

```
VAR
  fireStation : FireStation(district.fires, car1.going_back_ack,
                            car2.going_back_ack, car3.going_back_ack,
                            car4.going_back_ack, car5.going_back_ack);
  car1 : FireFightingCar(fireStation.car1_go);
  car2 : FireFightingCar(fireStation.car2_go);
  car3 : FireFightingCar(fireStation.car3_go);
  car4 : FireFightingCar(fireStation.car4_go);
  car5 : FireFightingCar(fireStation.car5_go);
  district : District(count(
                            next(car1.fire_extinguished_ack),
                            next(car2.fire_extinguished_ack),
                            next(car3.fire_extinguished_ack),
                            next(car4.fire_extinguished_ack),
                            next(car5.fire_extinguished_ack)
                           )
                     );
```

To check that the system works correctly we ask whether it is always able to eventually quench all fires:

```
LTLSPEC NAME eventually_extinguished :=
        G (district.fires > 0 -> F district.fires = 0);
```

The property is satisfied. However we cannot use it as the negation of our Top-Level Event because it is expressed in LTL and not as an invariant. Next section discusses how a watching monitor can be employed in cases like this to entail a weaker although still valuable result.

Model Extension and Fault Tree construction

In our specific use case, as one possible instance of system faults, cars will possibly disappear, simulating their unavailability. Cars might be unavailable because broken or because firemen themselves are not available. Details are not meaningful. Car unavailability will be modeled in the Fire Station failing to connect with the car to tell it to go.

Another fault that we will model will be the case of a car moving but never reaching to destination. An accident happening to the firefighting car might be cause of this. Finally we will model the case of a car losing connection with the Fire Station once the fire is quenched. This might be due to physical issues in the connection, like interference or such, and causes the car not to come back to the station (for a car to go back it should be given permission by the Fire Station).

Since Top-Level Events can only be expressed by invariant formulae in XSAP, we will need some workaround to guarantee that fires will always be extinguished, eventually. To do that we implement a monitor, which operates as a supervisor on the district. The purpose of the monitor is to let the system know when a fire is not extinguished in a fixed amount of time, by triggering a timeout variable after then, if fires still have not been quenched.

```
MODULE Monitor(fires, new_fires)
DEFINE MAX_TIME := 10; -- time before timeout from fire burst

-- Start counting from 0, trigger once you see fires and go
-- back to 0 once MAX_TIME is reached. If fires have not
-- been extinguished this will be detected.
VAR
  time_counter : 0..MAX_TIME;
ASSIGN
  init(time_counter) := 0;
  next(time_counter) := case
                          (time_counter=0) & (fires=0) : 0;
                          (time_counter=0) & (fires>0) :  -- triggered
                                (time_counter + 1) mod (MAX_TIME+1);
                         TRUE : (time_counter + 1) mod (MAX_TIME+1);
                        esac;

-- this models the final fire-free time window to extinguish the fire
DEFINE TIME_WINDOW := 4;
DEFINE
  fires_are_coming_late := (
               (time_counter > MAX_TIME - TIME_WINDOW)
                     &
               (new_fires>0)
           );
  they_came_late_already_anyway := ( (newFiresCameTooLate) & (fires>0)
      );

VAR newFiresCameTooLate : boolean;
ASSIGN
  init(newFiresCameTooLate) := FALSE;
  next(newFiresCameTooLate) := fires_are_coming_late |
                               they_came_late_already_anyway;

-- this defines the timeout for fires to be all quenched
DEFINE timeout := (time_counter=MAX_TIME) & (fires>0);
```

This will provide a bounded guarantee. For a reasonable analysis, we will demand new fires not to appear in the final time window before the timeout, otherwise the game would already be lost at the beginning. This, together with the timeout event not happening, is expressed in our Top-Level Event specification.

```
-- Definition of the good behavior: its negation is the TLE
INVARSPEC NAME TLE_negated :=
            monitor.timeout -> monitor.newFiresCameTooLate;
```

The code follows, modified with an account for the hook variables. We cut the duplication of code with respect to the previous presentation of modules. The district is not subject to faults in our model, therefore we omit its model specification completely.

```
MODULE FireStation(fires, car1_coming_back, car2_coming_back,
                    car3_coming_back, car4_coming_back,car5_coming_back)
IVAR
  fault_car1_unavailable_connection : boolean;
  fault_car2_unavailable_connection : boolean;
  fault_car3_unavailable_connection : boolean;
  fault_car4_unavailable_connection : boolean;
  fault_car5_unavailable_connection : boolean;

  car1_connection_back : boolean;
  car2_connection_back : boolean;
  car3_connection_back : boolean;
  car4_connection_back : boolean;
  car5_connection_back : boolean;

----------- little technicality -----------
VAR
  car1_coming_back__var : boolean;
  car2_coming_back__var : boolean;
  car3_coming_back__var : boolean;
  car4_coming_back__var : boolean;
  car5_coming_back__var : boolean;
ASSIGN
  car1_coming_back__var := car1_coming_back;
  car2_coming_back__var := car2_coming_back;
  car3_coming_back__var := car3_coming_back;
  car4_coming_back__var := car4_coming_back;
  car5_coming_back__var := car5_coming_back;
--------------------------------------------------

VAR
  car1_isHere : boolean;  car1_go : boolean;  car1_gone : boolean;
  car2_isHere : boolean;  car2_go : boolean;  car2_gone : boolean;
  car3_isHere : boolean;  car3_go : boolean;  car3_gone : boolean;
  car4_isHere : boolean;  car4_go : boolean;  car4_gone : boolean;
  car5_isHere : boolean;  car5_go : boolean;  car5_gone : boolean;

  ...
  ...
  ...

  next(car1_gone) := car1_go & next(!car1_coming_back__var);
  next(car2_gone) := car2_go & next(!car2_coming_back__var);
  next(car3_gone) := car3_go & next(!car3_coming_back__var);
  next(car4_gone) := car4_go & next(!car4_coming_back__var);
  next(car5_gone) := car5_go & next(!car5_coming_back__var);
```

```
MODULE FireFightingCar(sent_now)
IVAR
```

```
   fault_unavailable_car : boolean;
   fix_back_available : boolean;

VAR
   just_arrived : boolean;
   fire_extinguished_ack : boolean;
   going_back_ack : boolean;

ASSIGN
   init(just_arrived) := FALSE;
   next(just_arrived) := (sent_now=TRUE);

   ...
   ...
   ...

   init(going_back_ack) := FALSE;
   next(going_back_ack) := fire_extinguished_ack & !going_back_ack;
```

```
MODULE main
VAR
   fireStation : FireStation(district.fires, car1.going_back_ack,
                             car2.going_back_ack, car3.going_back_ack,
                             car4.going_back_ack, car5.going_back_ack);

   ...
   ...
   ...

   monitor : Monitor(district.fires, district.new_fires);
```

The extensions are on the FireFightingCar module and on the FireStation module: unavailabilities are modeled as connections stuck at inactive. The resulting file for the Fault Extension is rather long because it has to account for faults for all cars in the Fire Station module. We only report those relative to one car:

```
FAULT EXTENSION FE_CAE_ATG

-- Any car can may unavailable at any moment on the way to the fire
    place
-- (thus never arrive) --
  EXTENSION OF MODULE FireFightingCar
    SLICE Car_makeUnavailable AFFECTS just_arrived WITH
      MODE is_unavailable : Transient StuckAtByValue_I (
                        data      term << FALSE,
                        data      input << just_arrived,
                        data      varout >> just_arrived,
                      template    self_fix = fixed,
                        event     failure >> fault_unavailable_car,
                    event     fixed >> fix_back_available
       );

-- For simplicity we assume that it never happens that communications
-- are unavailable in either directions (base2car at the start of
-- actions and car2base when the fire is extinguished). That could
-- be obtained duplicating slices instead of duplicating modes.
  EXTENSION OF MODULE FireStation
    SLICE cannot_communicate_car1 AFFECTS car1_go ,
        car1_coming_back__var WITH
          MODE unreachable_car : Transient StuckAtByValue_I (
                            data      term << FALSE,
                            data      input << car1_go ,
                            data      varout >> car1_go ,
                           template  self_fix = fixed ,
                            event     failure >>
                                fault_car1_unavailable_connection ,
                            event     fixed >> car1_connection_back
                           );
         MODE unreachable_base : Transient StuckAtByValue_I (
                            data      term << FALSE,
                            data      input << car1_coming_back__var ,
                            data      varout >> car1_coming_back__var ,
                           template  self_fix = fixed ,
                            event     failure >>
                                fault_car1_unavailable_connection ,
                            event     fixed >> car1_connection_back
                          );

    SLICE cannot_communicate_car2 AFFECTS car2_go ,
        car2_coming_back__var WITH
     ...
     ...
```

The results are shown in figure 6.3. As we can see the tree finds 10 minimal cut sets. The 8 singletons on the right indicate that the failure of either one of the cars in $\{1, 2, 3, 4\}$ can lead to a failure of the desirable property. For the fifth car this is not enough (left view): the first car has to be non operative also, otherwise it could supply for $car5$ once back to the Fire Station. Notice that only one of two faults of $car1$ is expressed in the left view, namely the fault on the connection with the Fire Station from the car. Here all other ways to make $car1$ unavailable are already covered by minimality of the singleton cut sets on the right: this would be duplicated information that the fault tree construction procedure automatically excludes.

A note on performance

We have seen in section 4.2.2 that the base of XSAP is founded on algorithms of nuXmv. As indicated in section 4.3.2, there are four engines available for the Fault

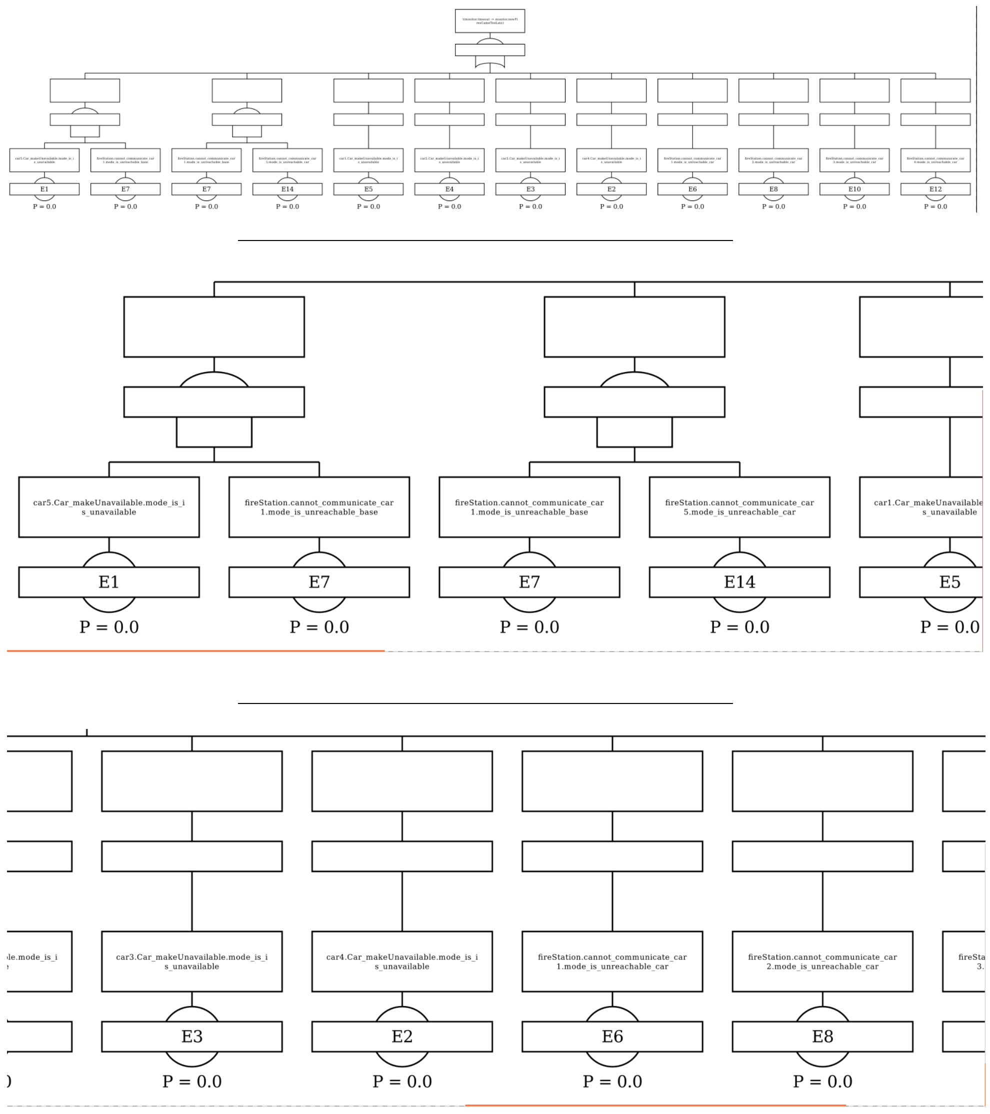

car5.Car_makeUnavailable.mode_is_i
s_unavailable
fireStation.cannot_communicate_car
1.mode_is_unreachable_base
fireStation.cannot_communicate_car
1.mode_is_unreachable_base
fireStation.cannot_communicate_car
5.mode_is_unreachable_car
car1.Car_makeUnavailable
s_unavailable
E1
E7
E7
E14
E5
P = 0.0
car3.Car_makeUnavailable.mode_is_i
s_unavailable
car4.Car_makeUnavailable.mode_is_i
s_unavailable
fireStation.cannot_communicate_car
1.mode_is_unreachable_car
fireStation.cannot_communicate_car
2.mode_is_unreachable_car
E3
E2
E6
E8
P = 0.0

Figure 6.3: The Fault Tree constructed by XSAP for the CAE model (with left view and central view).

Tree construction, namely **bdd**, **bmc**, **msat** and **ic3**. The CAE example is simple enough to let us highlight some aspects regarding performance without incurring into intricacies and model complexities as we had for the thermostat use case example. Moreover, the CAE example allows us to easily control performances by setting different values on parameters, such as `MAX_TIME`.

We show in figure 6.4 the different performance of XSAP using different engines, based on the bmc bound k. The `bdd` and `ic3` methods do not depend on the bound thus their performance are constant. Since we set `MAX_TIME`=10 in the monitor,

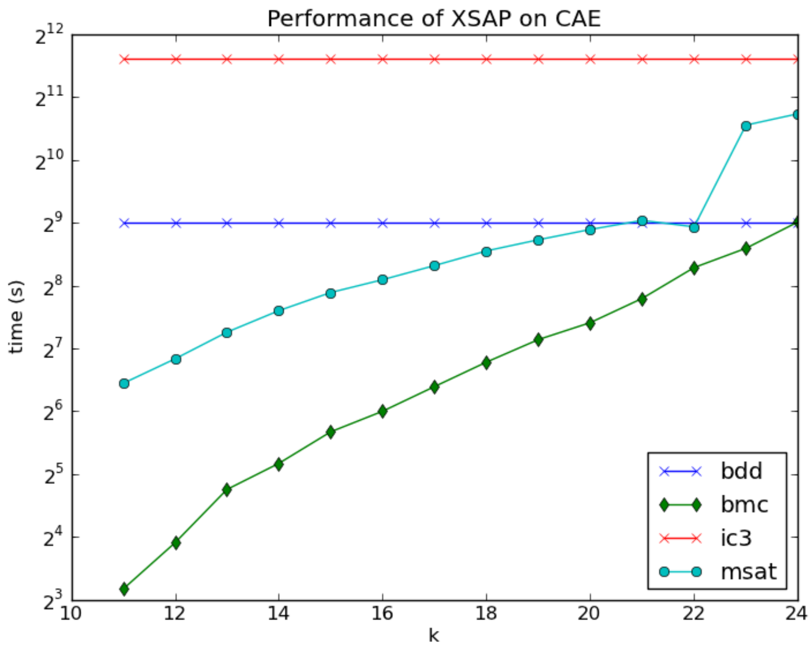


Figure 6.4: Performance (logarithmic scale) of XSAP on the CAE model using different engines.

the bmc procedure is inconclusive for bounds smaller than 11 (the first of 11 is initialization). The fastest algorithm up to $k = 24$ seems to be SAT-based `bmc`. `ic3` is the slowest on this model[6].

After the bound $k = 11$ we expect fault trees to be all equivalent, because the monitor is defined using the same `MAX_TIME`, which entails executions that all monotonically subsume to the first. This is the case indeed, therefore the consideration that we make in this little part have all a quantitative flavour. Notice that if we did not have domain knowledge about the system we couldn't claim satisfaction for any k at all. This delineates a trade-off between the use of bounded model checking and complete methods. For the CAE example, if less than $k = 23$ steps were not enough to hold confidence in the model, then it would be better to use the plain bdd procedure, because it would take less.

Unfortunately the cut-off value where choosing one method if preferable to the other cannot be known in advance thus the trade-off can exploit little or no black-box guidance. Consider for example that increasing `MAX_TIME` from 10 to 12 rises the computation time from 8 to 53 minutes, whereas the bmc computation time

[6]Please notice that the performance that we found for the different engines on the CAE are not absolute. For example we expect that on models demanding for theory reasoning, `msat` be best (or even the only available option on infinite systems).

setting `MAX_TIME`=23 goes from 8 to 14 minutes at $k = 24$. Similar to what is usual in model checking, a bmc procedure can be used in the earlier phases of development to find Fault Trees in a very fast way and think about completeness later for self-assurance. In the next section we will show that there are cases in which bdd reasoning can give a more compact resolution of faults, even needing less time than bmc procedures for the same results.

6.3 Positive-XSAP

The tool XSAP has been designed for the automatic construction of artifacts for the analysis of hazardous situations and, how we saw, the features it provides can be lifted with some thoughtful reconsiderations to large-scale systems based on service orientation. As an experimental assessment, in this section we will show how to exploit the features of the tool beyond that level of understanding, not directly for the dependability analysis but for design and planning.

6.3.1 Exploiting duality

From a functional perspective, the fault tree construction procedure of XSAP resolves a model checking problem over fault-extended models with invariants, finding counterexamples that end up being part, either explicitly or implicitly, of the fault tree. When constructing the fault tree it is instinctive to set the Top-Level Event as an unwanted hazard, nonetheless nothing prevents us to change on this convention and reuse the same construction utilities for other ends.

This reasoning brought us to an understanding of the tool in its dualized version, that we will call Positive-XSAP[7]. In Positive-XSAP the Top-Level Event is no longer a bad state, but a desirable one. The extension of the model will no longer be performed by introducing unintentional happenings, but welcoming positive events and letting them construct nice system configurations.

We will see the positive procedure applied to the CAE example. The question that we would like to answer is whether we can find a non-faulty configuration of cars that can manage over a maximum of 5 fires in less than 10 time units. To answer this question we need to lay out an empty system of one District, one Fire Station and no cars. This will be our nominal model. After that we will define the injections. What we inject in this case are not faults, but cars. We let cars appear to the scene and call this event a positive fault.

After launching the fault tree construction using as a Top-Level Event our desired formula, we will obtain all possible minimal cut sets possibly leading the system to the satisfaction of the desired formula. We will call the dual fault tree of Positive-XSAP a *suggestion tree*.

[7]To avoid any misunderstanding we underscore that Positive-XSAP is only a convenient way that we use to call the XSAP procedures applied in a dualized fashion: it is by no means a new tool.

Remark 2. It is important to notice that, since we only changed perspective on the functional attributes of XSAP, the same properties as before hold. In particular we can only feed invariants to Positive-XSAP for checking, and we will be returned with minimal cut sets bound to special counterexamples that are, at the end of the story, existential paths. As a consequence, in presence of non-determinism (such as the ignition of new fires), constructions belonging to the suggestion trees will not necessarily satisfy the desired property in a universal sense. This is the very reason why we called them suggestion trees. We will come back to this point in section 6.3.3.

6.3.2 Implementation

The nominal SMV model for Positive-XSAP is a plain CAE with no active cars. We model their appearance by means of boolean activation of cars. Activations correspond to boolean flag variables, always `FALSE` in nominal mode. Cars can possibly perform actions only if their corresponding activation variables are set to `TRUE` by the injection mechanism. Faults, namely activations, are permanent stuck-at `TRUE`.

The monitor specification is kept unchanged from the CAE system of the last section, as well as the District's, the FireFightingCar's and the main's. For this reason we only report the specification for the FireStation in the following:

```
MODULE FireStation(fires , car1_coming_back , car2_coming_back ,
                   car3_coming_back , car4_coming_back ,car5_coming_back)
IVAR
  fault_car1_available_connection : boolean;
  fault_car2_available_connection : boolean;
  fault_car3_available_connection : boolean;
  fault_car4_available_connection : boolean;
  fault_car5_available_connection : boolean;

---------- the activation section ----------
VAR
  car1_go_active : boolean;        car1_coming_back_active : boolean;
  car2_go_active : boolean;        car2_coming_back_active : boolean;
  car3_go_active : boolean;        car3_coming_back_active : boolean;
  car4_go_active : boolean;        car4_coming_back_active : boolean;
  car5_go_active : boolean;        car5_coming_back_active : boolean;

ASSIGN
  car1_go_active := FALSE;        car1_coming_back_active := FALSE;
  car2_go_active := FALSE;        car2_coming_back_active := FALSE;
  car3_go_active := FALSE;        car3_coming_back_active := FALSE;
  car4_go_active := FALSE;        car4_coming_back_active := FALSE;
  car5_go_active := FALSE;        car5_coming_back_active := FALSE;
--------------------------------------------------

VAR
VAR
```

```
  car1_isHere : boolean;   car1_go : boolean;   car1_gone : boolean;
  car2_isHere : boolean;   car2_go : boolean;   car2_gone : boolean;
  car3_isHere : boolean;   car3_go : boolean;   car3_gone : boolean;
  car4_isHere : boolean;   car4_go : boolean;   car4_gone : boolean;
  car5_isHere : boolean;   car5_go : boolean;   car5_gone : boolean;

DEFINE __going_cars__ := count(
                               car1_go | car1_gone,
                               car2_go | car2_gone,
                               car3_go | car3_gone,
                               car4_go | car4_gone,
                               car5_go | car5_gone
                              );

ASSIGN
  init(car1_isHere) := TRUE;        init(car1_go) := FALSE;
  init(car2_isHere) := TRUE;        init(car2_go) := FALSE;
  init(car3_isHere) := TRUE;        init(car3_go) := FALSE;
  init(car4_isHere) := TRUE;        init(car4_go) := FALSE;
  init(car5_isHere) := TRUE;        init(car5_go) := FALSE;

  next(car1_isHere) := (car1_coming_back & car1_coming_back_active)
                     | (car1_isHere & !car1_go);
  next(car2_isHere) := (car2_coming_back & car2_coming_back_active)
                     | (car2_isHere & !car2_go);
  next(car3_isHere) := (car3_coming_back & car3_coming_back_active)
                     | (car3_isHere & !car3_go);
  next(car4_isHere) := (car4_coming_back & car4_coming_back_active)
                     | (car4_isHere & !car4_go);
  next(car5_isHere) := (car5_coming_back & car5_coming_back_active)
                     | (car5_isHere & !car5_go);

  next(car1_go) := car1_go_active & (fires - __going_cars__ > 0)
                                  & next(car1_isHere);
  next(car2_go) := car2_go_active & (fires - __going_cars__
                                      - count(next(car1_isHere)) > 0)
                                  & next(car2_isHere);
  next(car3_go) := car3_go_active & (fires - __going_cars__
                                      - count(next(car1_isHere),
                                              next(car2_isHere)) > 0)
                                  & next(car3_isHere);
  next(car4_go) := car4_go_active & (fires - __going_cars__
                                      - count(next(car1_isHere),
                                              next(car2_isHere),
                                              next(car3_isHere)) > 0)
                                  & next(car4_isHere);
  next(car5_go) := car5_go_active & (fires - __going_cars__
                                      - count(next(car1_isHere),
                                                                  next(car2_isHere),
                                                                  next(car3_isHere),
                                                                  next(car4_isHere)) > 0)
                                                    & next(car5_isHere);

  next(car1_gone) := car1_go & next(!car1_coming_back);
  next(car2_gone) := car2_go & next(!car2_coming_back);
  next(car3_gone) := car3_go & next(!car3_coming_back);
  next(car4_gone) := car4_go & next(!car4_coming_back);
  next(car5_gone) := car5_go & next(!car5_coming_back);
```

The actual injection is really simple: permanently decide some of the activation variables to TRUE. We report the injection on the first car only: it is no different for all the other 5. Notice that, unlike the injection for the CAE that we saw in the last section, here activations might happen concurrently. Consequently we separate each mode in different slices.

```
FAULT EXTENSION FE_CAE_ATG
   EXTENSION OF MODULE FireStation
      SLICE car1_s1 AFFECTS car1_go_active WITH
            MODE reachable_car: Permanent StuckAtByValue_I (
                  data     term << TRUE,
                  data     input << car1_go_active,
                  data     varout >> car1_go_active,
                  event    failure >> fault_car1_available_connection
                 );
      SLICE car1_s2 AFFECTS car1_coming_back_active WITH
           MODE reachable_base: Permanent StuckAtByValue_I (
                  data     term << TRUE,
                  data     input << car1_coming_back_active,
                  data     varout >> car1_coming_back_active,
                  event    failure >> fault_car1_available_connection
                );

      SLICE car2_s1 AFFECTS car2_coming_back_active WITH
                  ...
                  ...
                  ...
```

A special care has to be given to the Top-Level Event specification, in this case. As we discussed above, the result of Positive-XSAP is not a construction granting the Top-Level desirable property to hold universally, only existentially. This means that to every variable whose value is not deterministically determined, it will be choice for the tool. In other words, the tool builds suggestion trees by responding to the call of picking arbitrary variable values so to reach the desirable property.

Our plain Top-Level event would be (notice the upfront negation, due to Positive-XSAP dualization):

```
INVARSPEC NAME TLE_negated :=
            !(!monitor.timeout & !monitor.newFiresCameTooLate);
```

This specification is not fine to our aims because Positive-XSAP is allowed to pick any arbitrary value for its variables, and to be minimal it will centrainly prefer those configurations with no fires ever bursting out in the scene. For that case, with that Top-Level specification, suggestion trees of Positive-XSAP will be of no use.

Therefore we will need to enforce some sort of fairness to the construction of the suggestion tree, by acting on the property specification. The way to do it is by asking that in the final state — corresponding to the property violation — the district experience all the possible fire explosions and that the monitor reach a time count of `MAX_TIME` (i.e. our maximum before timeout):

```
INVARSPEC NAME TLE_negated :=
            !(
               district.newfire_counter=district.MAX_FIRES
              & monitor.time_counter=MAX_TIME
              & !monitor.timeout & !monitor.newFiresCameTooLate
             );
```

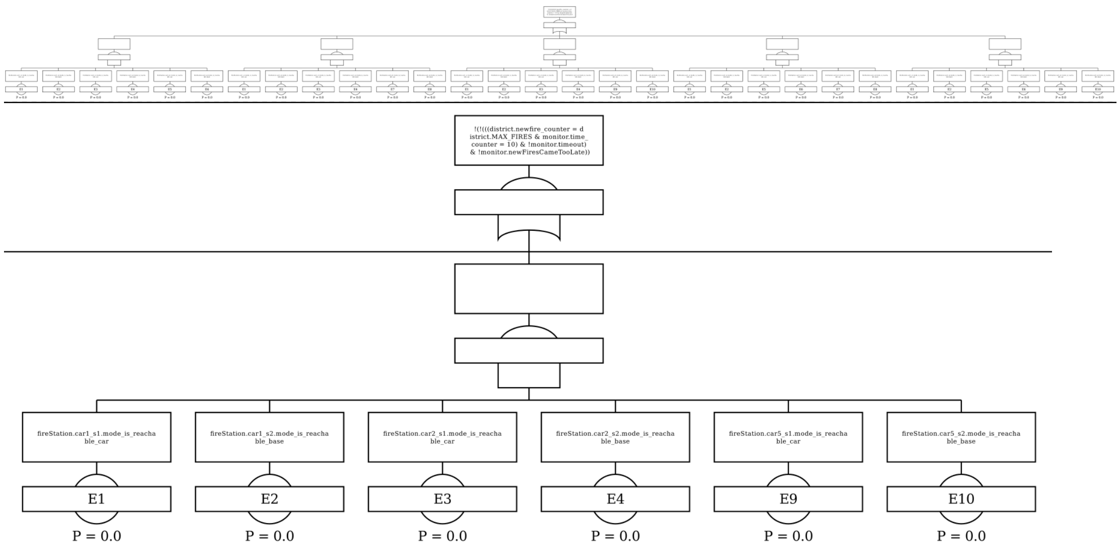

!(!(((district.newfire_counter = d
istrict.MAX_FIRES & monitor.time_
counter = 10) & !monitor.timeout)
& !monitor.newFiresCameTooLate))
fireStation.car1_s1.mode_is_reacha
ble_car
fireStation.car1_s2.mode_is_reacha
ble_base
fireStation.car2_s1.mode_is_reacha
ble_car
fireStation.car2_s2.mode_is_reacha
ble_base
fireStation.car5_s1.mode_is_reacha
ble_car
fireStation.car5_s2.mode_is_reacha
ble_base
E1
E2
E3
E4
E9
E10
P = 0.0

Figure 6.5: The Fault Tree constructed by Positive-XSAP for the CAE model (with zoom on the middle part).

Now the construction is fine and ends up in 4 minimal cut sets, that we show in figure 6.5. All the cut sets are equivalent and all resemble the middle one, on which we zoomed in the figure.

From the picture we acknowledge that there is a particular configuration that leads all the fires to be extinguished as required. In particular, we need any two cars among those in $\{2, 3, 4, 5\}$, arbitrarily, based on when fires burst plus car $car1$. For example the displayed configuration in the middle, despite not much visible, has the configuration $\{1, 2, 5\}$. Car $car1$ is special and needs to be there in all minimal configurations: since failures are not admitted, all cars in $\{2, 3, 4, 5\}$ are instructed to `go` only if $car1$ is not there at the Fire Station (the reader might want to check this with the code).

Considerations about scheduling of cars can be made by looking at the traces produced by the tool, which contain event ordering information: Positive-XSAP, for our example, produced 5, that we decided not to present them here to avoid visual cluttering. The time for the explosion of new fires was set arbitrarily by XSAP, not always with the same pattern. We would like to stress that one could possibly push even stricter requirements to the system, feasibly controlling the entire non-determinism from the desirable Top-Level Event formula and achieve more specialized trees. We decided not to do that because our study is limited to comprehension.

The suggestion tree of figure 6.5 could be found using bmc methods with bound $k = 11$. It took $3.173s$ using the SAT-based `bmc` procedure and $25.812s$ using `msat`. Then we tried to feed the problem to the `bdd` and `ic3` engines. Not without surprise XSAP was able to find smaller cut sets using those procedures, respectively in $4.668s$ and $38.069s$. Results are shown in figure 6.6. The reason for this new, different suggestion tree lays in the completeness of the `bdd` and `ic3` methods: XSAP can look at the whole state space and explore more options to make the existential paths minimal.

After an exploration of the produced trace we were able to grasp that a path could be generated such that the single car $car1$ could make its way back and forth from the FireStation to extinguish all fires independently, in 24 steps. The same fault tree could then be find using bmc-based techniques using, this time, the bound $k = 24$. This time, however, it took much more time: around $16s$ for the `bmc` procedure and more than $300s$ for `msat`.

6.3.3 Suggestion trees, SOA and planning tableaux

We mentioned at the end of section 6.3.1 that suggestion trees cannot be taken as globally universal design utilities, mainly because their construction is inherently existential on traces. However, although lacking universality, they might come to suggest interesting paths of exploration for design and might come in handy when system reconfigurations are required.

The first short part hereafter is dedicated to the discussion of this aspect with respect to the paradigm of service-orientation. After that we show here how the procedure is not limited to design but can be employed to other ends, such as planning. This, with relative comments, will conclude the section.

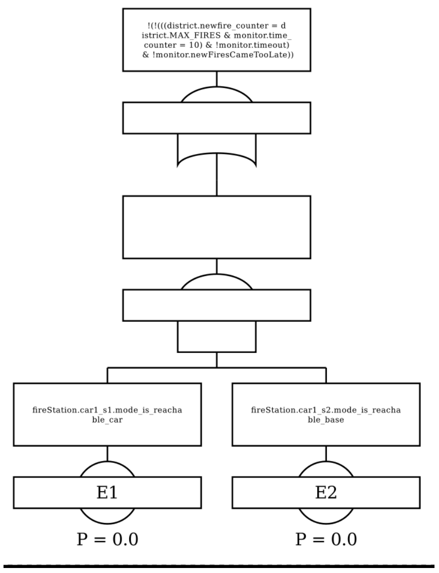

!(!(((district.newfire_counter = district.MAX_FIRES & monitor.time_counter = 10) & !monitor.timeout) & !monitor.newFiresCameTooLate))
fireStation.car1_s1.mode_is_reachable_car
fireStation.car1_s2.mode_is_reachable_base
E1
E2
P = 0.0
P = 0.0

Figure 6.6: The Fault Tree constructed by Positive-XSAP (BDD and IC3).

Positive-XSAP for SOA design

Service-orientation is particularly sensible to the problem of prompt system reconfiguration. In [61] is proposed a structuring of SOA that is based on an interface filter bound to feed and get fed by an underlying component. We discussed about this formal architecture in section 2.2.2 where we identified three different *modules* for that, namely the functional, interface and interface factory module. In a contract-based vision of the system, we can identify the functional module with implementations, the interface module with contracts and the factory module with the dynamic system adaptations that can happen offline, in redesigning the system.

This re-adaptation, which we referred to as *dynamic system reconfiguration* in section 2.3, might be based on suggestion trees of XSAP to decide which of the interface components to lay out and which parameters to use. The significant thing to be aware of is the existential aspect of XSAP, which prevents the provision of guarantees in general and the employment of FMEA tables might help on this. The whole thing would need a great deal of exploration, that we consciously leave to future work.

Positive-XSAP for planning

On a similar line we can notice that the same method can be used to resolve a planning problem under the assumption of non-determinism. If the previous problem asked how the given resources could be arranged in the system in order to reach the specified property, the planning question asks how a certain goal can be reached given fixed conditions. Coincidentally or by accident the Positive-XSAP problem over CAE can be restated as an instance of both, depending on the objective given to the analysis.

Under the design perspective, CAE wants to find a design that allows every fire to be quenched before the timeout is triggered. To feel this more concretely, imagine a new district is assigned to the Fire Station. Given a suitable redefinition of the model, it is asked at design to provide the minimum number of cars to cover fires of both districts and avoid timeout. This view follows what has been said so far and suffers from the issues of existentiality vs. universality that we previously mentioned, especially if too much freedom is given to non-determinism.

In the eyes of a planner, instead, what one looks for is really a way, *any* way, to reach a given goal. Given 5 available cars and fires bursting at predefined times, the planner asks for the minimal configurations of cars that can allow them to be all extinguished before timeout. Possibly the result of the analysis with Positive-XSAP will return a fault tree of possible options. We call the fault tree generated by Positive-XSAP for planning a *planning tableau*, because it shows more than one way, all minimal, for the planner to choose its strategy and reach its goal. The nicest thing about planning tableaux comes from the fact that enabling option `--gen-trace` in the Positive-XSAP process, a trace description is attached to each set, which specifies precisely the way to reach the goal in the minimal sense.

On the employment of Positive-XSAP

The approaches that we put forward in this section are both novel and experimental, since they have no direct correspondence in the literature to the extent that service-orientation is concerned. However, similar approaches have been devised accounting for planning using model checking techniques. For example in [31] the problem of conformant planning in non-deterministic domains is tackled. This problem aims at finding paths in a given domain of actions that can lead an actor to achieve a goal regardless of non-determinism. In the paper this is done, symbolically, evolving the system by means of a transition relation between indistinguishable sets of states, called *belief states* with a not-so-cryptic reference to epistemic logic and epistemic reasoning.

The work in [31] is only one examples of planning strategies that the model checking community has developed throughout the years. Our approach is algorithmically similar but methodically different. Here we proposed *suggestion trees* and *planning tableaux* for different aims in the SOA domain, as an integration, a tool exploitation, to classical and non classical dependability analysis techniques. From this new standpoint we believe that industry might widely benefit from the adoption of automatic tools like XSAP, which can find application in more than one context, to assess properties beyond the yet always central system dependability.

Final Considerations and Future Work

The problem of assessing dependability of SOAs in terms of Fault Tree Analysis (FTA), Failure Mode and Effects Analysis (FMEA) or related techniques has never received much attention by the research community, because of their original statement in the context of Web services. Only recently the literature has started to appreciate the employment of SOA as an infrastructure for developing Cyber-Physical Systems; works relating the understanding of faults occurring in this context are becoming of greater and greater value and actually starting to get investigated [4].

It his thesis we presented novel ways to assess the dependability of such complex Cyber-Physical Systems constructed over Service Oriented Architectures. Dependability assessment was proposed through Model-Based Fault Injection put in place with the features brought by the XSAP safety analysis tool, which is based on state-of-the-art symbolic model checking techniques.

We saw how service orientation readily fits into the contract framework, which is a mathematical basis purposely created to track component interactions at the interface level, accounting for feasibility of embedding in the assumptions and enforcing property to the environment in the guarantees. We provided an overview of the languages employed in the description of SOA and Cyber-Physical Systems in general, ending up preferring SysML+SoaML, as a standard UML derivation, for their amenability to industry needs.

Interestingly, in the literature context our work is to be placed at a point of intersection between the SOA literature, the Cyber-Physical Systems literature, the formal-verification literature and industrial based software engineering practice. This lead us to a cross-cutting treatment, taking useful features from one domain or the other, disposing of the irrelevant ones. This was done thoughtfully after a deep analysis of each aspect.

In the process of doing that we proposed how to use the XSAP tool with a good deal of novelty and adaptation to the SOA realm. We proposed the introduction of latency ports for timings, inspired by [15] and similarly adaptable to other non-functional aspects. Here we wanted to show how the timing viewpoint could be made explicit and how faults could be injected on them by means of XSAP, in spite of it not having been conceived to work out latencies as injections. We showed how the topological evolution could be tracked by means of fault injection using boolean availability techniques and even how we could rethink the fault tree construction of XSAP in its dualized fashion, promoting the concepts of suggestion trees and planning tableaux.

Several works are envisioned for the future. In the second part of the thesis we analyzed the use case instances of two different kind of SOA. For the first is representative the Thermostat example, for the other the CAE. Even though we presented the two examples distinctly, they do not need to be. The separation of treatments has been deemed convenient for presentation purposes but their integration, we believe, would be more than welcome upon expansion of our techniques to broader-spectrum industrial-size use cases that concomitantly treat service availability and system dynamics. Nicely, we claim, functional dynamic aspects can be confined to components' description while boolean availability and timing (as well as possibly other non-functional considerations) can be delegated to the network agent. This would have the prime advantage of having component's functional aspects treated on a different layer than non-functional and network-related aspects, thereby independently. This would be answer to the concern of adaptable embeddability for this kind of systems — or rather for their modeling aspect with respect to verification and dependability assessment — retaining the functional description of single components unvaried for all analysis where they are employed, only changing parameters of the environment where they are embedded, virtually including other factors besides the service-oriented network that we talked about.

Related to this we also saw how a contract-based description of systems can help against scalability issues, especially when feedback is involved. We saw for the Thermostat example that the relative contract-based dependability analysis was way faster than the direct analysis over the whole state space of the system, provided a prior resolution of the unfolded parallel composition of the system description. The state space for the single contract of the component, the Thermocouple in our case, included possibilities that never could have happened in practice but we argued that this feature, an abstraction of the possible states encompassing an over-approximation, was an expected activity in the view of proving contracts coherent, organically, with any environment they could embed into. Notice that in systems with control such as those employed in industry, this aspect takes solid concrete meaning.

The natural continuation of our work is a tool at the service of industry for the automatic translation between modeling standards and the input language for XSAP. We envision two realization possibilities. The first is the translation starting from our recommended modeling standard SysML+SoaML (see section 5.1), the other founds on dependability analyses based on different translation tools towards the safety platform. Both cases would imply a methodological study about the semantic transformations applicable on the language entities, provided that a semantics is available, and a special accounting for the SOA aspect.

The workflow that we propose for the first case is depicted in figure 6.7. If this op-

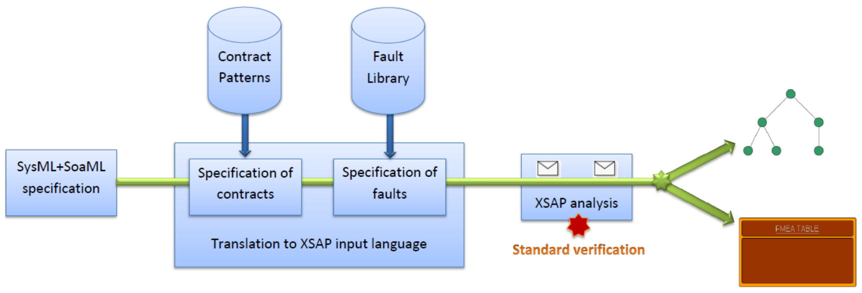


Figure 6.7: Future work: Final translation and utilization workflow

tion is preferred then it would be necessary to fix a contract-based language for the specification of system properties, such as BCL [52], and define how that could be supplied on the language entities. Moreover, it would be necessary to understand how the specification of faults could be attained in this context, because their specification in the form of injection files (such as the `.fei` file of XSAP) is as unfeasible as a direct modeling in nuXmv for the system description. As suggested in chapter 1 after considerations from the literature, we we could give the user a fixed library of injectable faults and supply it with the possibility of constructing slices for each entity of the language, choosing modes from a default shortlist[8]. Positive-XSAP is not meant to enjoy the same level of ease, even though attempts could be made towards that direction.

Note the red star in the workflow of figure 6.7: this is meant to represent the availability of nuXmv as a basis for verification in XSAP, once the nominal system model is available. Having a tool like XSAP, that comprises verification and dependability aspects under the same umbrella, is both convenient and important. In fact, the point becomes tangibly interesting at this point, where the proposed translation from SysML+SoaML produces the intermediate files to feed XSAP with, distinguishing the nominal model from the `.fei` file (the two envelopes in the figure). After that, at the end of the procedure, if not all requirements are met or the system is considered not ready for the following engineering phase, design can be reiterated knowledgeable of feedback from both the verification and the safety analysis underpinning.

The second option would be to approach the problem in a bottom-up Ptolemy-like fashion[67], where single components and viewpoints are independently developed using different modeling languages and then combined for the verification/dependability analysis phase. This approach, although certainly attainable, would need translation support towards injection by each modeling tool, which is a demanding request. Moreover the description of the service architecture would demand specialized construct which, for standard modeling language, are not built

[8]These might include boolean availability, time degradation or be based on functional properties. Throughout the implementations of chapter 6 we only made use of *stuck-at* faults for our injections, despite XSAP provides a handful of more default options and the possibility of defining new[54].

in. SysML+SoaML, on the contrary, separates architecture and functionality by design.

Feasibility of translation from UML-like languages to verification engines is witnessed by works such as [53] and [11]. Quite interestingly, there even exists an independent research work that translates SOA processes into SMV language[59]. In this work the translation starts from a WS-BPEL specification that, as we saw in section 5.1, is procedural and lacks some of the functionalities that we would need in our Cyber-Physical scenario. It is nevertheless remarkable to see that our projections to future work are aligned to other precedently developed methodologies in the literature, and that the means we are dealing with are too.

In conclusion, our methodology let us investigate interesting aspects of complex Cyber-Physical Systems based on SOA, from which we induced modeling practices whose adoption criteria we motivated in detail. The study put the basis on methods for a far-reaching treatment of those systems in terms of safety analysis and design, that we ended in an industrial level workflow proposal in this last chapter.

Performance derived from the safety analysis platform might still hinder the effectiveness of our approach. To date we yet do not have measures based on real industrial scenarios — which we expect way bigger and more complex — however we are confident that the contract-based approach that we proposed for the Thermostat use case could evenly be able to tackle the scalability challenge over more complex system models and that the boolean availability of components could be handled nicely on large scales by exploiting symbolic model checking and reachability procedures such as BDD, BMC or IC3 (cf. section 4.2.2). Eventually we expect that XSAP take complete advantage of the integrated SMT-solving possibilities of nuXmv, so that it can handle infinite state systems without need of discretization and let our analysis – ultimately our translation – be smoother and possibly faster.

Bibliography

[1] Aadl — http://www.aadl.info/.

[2] S. Abdelwahed, G. Karsai, N. Mahadevan, and S.C. Ofsthun. Practical implementation of diagnosis systems using timed failure propagation graph models. *Instrumentation and Measurement, IEEE Transactions on*, 58(2):240–247, Feb 2009.

[3] A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the linux kernel. In *Dependable Systems and Networks, 2004 International Conference on*, pages 867–876, June 2004.

[4] Cristiana Areias, Nuno Antunes, and JoãoCarlos Cunha. On applying fmea to soas: A proposal and open challenges. In István Majzik and Marco Vieira, editors, *Software Engineering for Resilient Systems*, volume 8785 of *Lecture Notes in Computer Science*, pages 86–100. Springer International Publishing, 2014.

[5] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. In *SPIN'06: Proceedings of the 13th international conference on Model Checking Software*, pages 146–162, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] M. Arrott, B. Demchak, V. Ermagan, C. Farcas, E. Farcas, I.H. Kruger, and M. Menarini. Rich services: The integration piece of the soa puzzle. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 176–183, July 2007.

[7] Karl Johan Aström and Richard M Murray. *Feedback systems: an introduction for scientists and engineers.* Princeton university press, 2010.

[8] Gilles Audemard, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. Bounded model checking for timed systems. In *Proceedings of the 22Nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, FORTE '02, pages 243–259, London, UK, UK, 2002. Springer-Verlag.

[9] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[10] Maurice Beek, Antonio Bucchiarone, and Stefania Gnesi. A survey on service composition approaches: From industrial standards to formal methods. In *In Technical Report 2006TR-15, Istituto*, pages 15–20. IEEE CS Press, 2006.

[11] Adrian Beer, Uwe Kühne, Florian Leitner-Fischer, Stefan Leue, and Rüdiger Prem. Quantitative safety analysis of non-deterministic system architectures. Technical report, 2013.

[12] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Formal methods for components and objects. chapter Multiple Viewpoint Contract-Based Specification and Design, pages 200–225. Springer-Verlag, Berlin, Heidelberg, 2008.

[13] Albert Benveniste, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim G. Larsen. Contracts for System Design. Research Report RR-8147, November 2012.

[14] Albert Benveniste, Benoit Caillaud, and Roberto Passerone. A Generic Model of Contracts for Embedded Systems. Research Report RR-6214, 2007.

[15] Albert Benveniste, Benoıt Caillaud, and Roberto Passerone. Multi-viewpoint state machines for rich component models. 2009.

[16] Albert Benveniste, Dejan Nickovic, and Thomas Henzinger. Compositional Contract Abstraction for System Design. Research Report RR-8460, January 2014.

[17] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W.Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 1999.

[18] Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. nuxmv 1.0 user manual. 2014.

[19] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, dependability and performance analysis of extended aadl models. *Comput. J.*, 54(5):754–775, May 2011.

[20] Marco Bozzano, Alessandro Cimatti, Cristian Mattarei, and Stefano Tonetta. Formal safety assessment via contract-based design. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis*, volume 8837 of *Lecture Notes in Computer Science*, pages 81–97. Springer International Publishing, 2014.

[21] Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. Symbolic fault tree analysis for reactive systems. In KedarS. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Automated Technology for Verification and Analysis*, volume 4762 of *Lecture Notes in Computer Science*, pages 162–176. Springer Berlin Heidelberg, 2007.

[22] Marco Bozzano and Adolfo Villafiorita. Improving system reliability via model checking: The fsap/nusmv-sa safety analysis platform. In Stuart Anderson, Massimo Felici, and Bev Littlewood, editors, *Computer Safety, Reliability, and Security*, volume 2788 of *Lecture Notes in Computer Science*, pages 49–62. Springer Berlin Heidelberg, 2003.

[23] Marco Bozzano and Adolfo Villafiorita. The fsap/nusmv-sa safety analysis platform. *International Journal on Software Tools for Technology Transfer*, 9(1):5–24, 2007.

[24] Marco Bozzano and Adolfo Villafiorita. *Design and safety assessment of critical systems.* CRC Press, 2010.

[25] Bpmn — http://www.omg.org/spec/bpmn/.

[26] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1), February 2007.

[27] S. Bruning, S. Weissleder, and M. Malek. A fault taxonomy for service-oriented architecture. In *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, pages 367–368, Nov 2007.

[28] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, Aug 1986.

[29] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, Jun 1990.

[30] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification*, pages 334–342. Springer International Publishing, 2014.

[31] A. Cimatti, M. Roveri, and P. Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1–2):127 – 206, 2004.

[32] Alessandro Cimatti. Industrial applications of model checking. In Franck Cassez, Claude Jard, Brigitte Rozoy, and MarkDermot Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 153–168. Springer Berlin Heidelberg, 2001.

[33] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and KimGuldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer Berlin Heidelberg, 2002.

[34] Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 277–293, Berlin, Heidelberg, 2012. Springer-Verlag.

[35] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 46–61. Springer, 2014.

[36] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[37] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *IN: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON THEORY AND APPLICATIONS OF SATISFIABILITY TESTING (SAT-2007*, pages 334–339, 2007.

[38] Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. Requirements validation for hybrid systems. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 188–203. Springer Berlin Heidelberg, 2009.

[39] Alessandro Cimatti and Stefano Tonetta. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming*, 97, Part 3(0):333 – 348, 2015. Object-Oriented Programming and Systems (OOPS 2010) Modeling and Analysis of Compositional Software (papers from {EUROMICRO} SEAA'12).

[40] D. Controneo, C. di Flora, and S. Russo. Improving dependability of service oriented architectures for pervasive computing. In *Object-Oriented Real-Time Dependable Systems, 2003. (WORDS 2003). Proceedings of the Eighth International Workshop on*, pages 74–81, Jan 2003.

[41] D. Cotroneo and R. Natella. Fault injection for software certification. *Security Privacy, IEEE*, 11(4):38–45, July 2013.

[42] O. Coudert and J.C. Madre. Fault tree analysis: 1020 prime implicants and beyond. In *Reliability and Maintainability Symposium, 1993. Proceedings., Annual*, pages 240–245, Jan 1993.

[43] Danse project — http://www.danse-ip.eu/home/.

[44] Danse deliverable d3.3 - concept alignment example description (publicly available).

[45] Luca de Alfaro and ThomasA. Henzinger. Interface theories for component-based design. In ThomasA. Henzinger and ChristophM. Kirsch, editors, *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer Berlin Heidelberg, 2001.

[46] Nicolas Dulac and Nancy Leveson. An approach to design for safety in complex systems. In *INT. SYMPOSIUM ON SYSTEMS ENGINEERING (INCOSE*, pages 33–407, 2004.

[47] Vina Ermagan, Claudiu Farcas, Emilia Farcas, Ingolf H. Krüger, and Massimiliano Menarini. A service-oriented approach to failure management. In Giese et al. [55], pages 102–116.

[48] Vina Ermagan, To-Ju Huang, Ingolf H Krüger, Michael Meisinger, Massimiliano Menarini, and Praveen Moorthy. Towards tool support for service-oriented development of embedded automotive systems. In *MBEES*, pages 1–24, 2007.

[49] Jonathan Ezekiel and Alessio Lomuscio. Combining fault injection and model checking to verify fault tolerance in multi-agent systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '09, pages 113–120, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[50] Jonathan Ezekiel and Alessio Lomuscio. A methodology for automatic diagnosability analysis. In *Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering*, ICFEM'10, pages 549–564, Berlin, Heidelberg, 2010. Springer-Verlag.

[51] Jonathan Ezekiel, Alessio Lomuscio, Levente Molnar, Sandor Veres, and Miles Pebody. Verifying fault tolerance and self-diagnosability of an autonomous underwater vehicle, 2011.

[52] O. Ferrante, R. Passerone, A. Ferrari, L. Mangeruca, and C. Sofronis. Bcl: A compositional contract language for embedded systems. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–6, Sept 2014.

[53] Alberto Ferrari, L Mangeruca, O Ferrante, and A Mignogna. Desyreml: a sysml profile for heterogeneous embedded systems. *Embedded Real Time Software and Systems, ERTS*, 2012.

[54] Embedded Systems Unit Fondazione Bruno Kessler. Xsap user manual, 2012.

[55] Holger Giese, Michaela Huhn, Ulrich Nickel, and Bernhard Schätz, editors. *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV, Schloss Dagstuhl, Germany, 7.-9. April 2008, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, volume 2008-2 of *Informatik-Bericht*. TU Braunschweig, Institut für Software Systems Engineering, 2008.

[56] Ibrahim Habli, Abdulaziz Al-Humam, Tim Kelly, and Leila Fahel. Integrating Safety Assessment into the Design of Healthcare Service-Oriented Architectures. In Volker Turau, Marta Kwiatkowska, Rahul Mangharam, and Christoph Weyer, editors, *5th Workshop on Medical Cyber-Physical Systems*, volume 36 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[57] Luke Thomas Herbert and Robin Sharp. *Workflow Fault Tree Generation Through Model Checking*, pages 2229–2236. C R C Press LLC, 2014.

[58] Matjaz B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2Nd Edition*. Packt Publishing, 2006.

[59] R. Kazhamiakin, P. Pandya, and M. Pistore. Timed modelling and analysis in web service compositions. In *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, pages 7 pp.–, April 2006.

[60] M. Kooli and G. Di Natale. A survey on simulation-based fault injection tools for complex systems. In *Design Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On*, pages 1–6, May 2014.

[61] Ingolf H. Krüger. From requirements to hierarchical soa models of cyberphysical systems. unpublished.

[62] Ingolf H. Krüger. Specifying services with {UML} and uml-rt: Foundations, challenges and limitations. *Electronic Notes in Theoretical Computer Science*, 65(7):34 – 50, 2002. {VISS} 2002, Validation and Implementation of Scenario-based Specifications (Satellite Event of {ETAPS} 2002).

[63] I.H. Kruger and R. Mathew. Systematic development and exploration of service-oriented software architectures. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 177–187, June 2004.

[64] Anna Lanzaro, Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. An empirical study of injected versus actual interface errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 397–408, New York, NY, USA, 2014. ACM.

[65] Rubén Lara, Dumitru Roman, Axel Polleres, and Dieter Fensel. A conceptual comparison of wsmo and owl-s. In Liang-Jie(LJ) Zhang and Mario Jeckle, editors, *Web Services*, volume 3250 of *Lecture Notes in Computer Science*, pages 254–269. Springer Berlin Heidelberg, 2004.

[66] Daniel Larsson and Reiner Hähnle. Symbolic fault injection, 2006.

[67] Edward A Lee and Stavros Tripakis. Modal models in ptolemy. In *EOOLT*, pages 11–21. Citeseer, 2010.

[68] Nancy G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety (Engineering Systems)*. The MIT Press, January 2012.

[69] Scott R. Little. *Efficient Modeling and Verification of Analog/Mixed-signal Circuits Using Labeled Hybrid Petri Nets*. PhD thesis, University of Utah, Salt Lake City, UT, USA, 2008. AAI3333598.

[70] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. Mcmas: A model checker for the verification of multi-agent systems. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 682–688. Springer Berlin Heidelberg, 2009.

[71] N. Looker, M. Munro, and Jie Xu. A comparison of network level fault injection with code insertion. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 1, pages 479–484 Vol. 2, July 2005.

[72] N. Looker and Jie Xu. Assessing the dependability of soap rpc-based web services by fault injection. In *Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall. The Ninth IEEE International Workshop on*, pages 163–163, Oct 2003.

[73] Nik Looker, Malcolm Munro, and Jie Xu. Assessing web service quality of service with fault injection. *Quality of Service for Application Servers, SRDS, Brazil*, 2004.

[74] Marcos López-Sanz, César J. Acuña, Carlos E. Cuesta, and Esperanza Marcos. Modelling of service-oriented architectures with {UML}. *Electronic Notes in Theoretical Computer Science*, 194(4):23 – 37, 2008. Proceedings of the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2007).

[75] P.D. Marinescu and G. Candea. Lfi: A practical and general library-level fault injector. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 379–388, June 2009.

[76] P. Nuzzo, A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli. Are interface theories equivalent to contract theories? In *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, pages 104–113, Oct 2014.

[77] Owl-s — http://www.w3.org/submission/owl-s/.

[78] R. Passerone, W. Damm, I. Ben Hafaiedh, S. Graf, A. Ferrari, L. Mangeruca, A. Benveniste, B. Josko, T. Peikenkamp, D. Cancila, A. Cuccuru, S. Gerard, F. Terrier, and A. Sangiovanni-Vincentelli. Metamodels in europe: Languages, tools, and applications. *Design Test of Computers, IEEE*, 26(3):38–53, May 2009.

[79] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.

[80] Antoine Rauzy. New algorithms for fault trees analysis. *Reliability Engineering & System Safety*, 40(3):203–211, 1993.

[81] M.Q. Saleem, J. Jaafar, and M.F. Hassan. Security modelling along business process model of soa systems using modified uml-soa-sec. In *Computer Information Science (ICCIS), 2012 International Conference on*, volume 2, pages 880–884, June 2012.

[82] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems*. *European Journal of Control*, 18(3):217 – 238, 2012.

[83] Soaml — http://www.omg.org/spec/soaml/.

[84] Soaml wiki — http://forge.modelio.org/projects/soaml-modelio3-user-manual-english/wiki.

[85] Sysml — http://www.omgsysml.org/.

[86] Robert A. Thacker, Kevin R. Jones, Chris J. Myers, and Hao Zheng. Automatic abstraction for verification of cyber-physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 12–21, New York, NY, USA, 2010. ACM.

[87] S.G. Vadlamudi and P.P. Chakrabarti. Robustness analysis of embedded control systems with respect to signal perturbations: Finding minimal counterexamples using fault injection. *Dependable and Secure Computing, IEEE Transactions on*, 11(1):45–58, Jan 2014.

[88] J. Voas. A tutorial on software fault injection.

[89] M Vollmer. Newton's law of cooling revisited. *European Journal of Physics*, 30(5):1063, 2009.

[90] Peng Wang, Yang Xiang, and Shao Hua Zhang. Cyber-physical system components composition analysis and formal verification based on service-oriented architecture. In *e-Business Engineering (ICEBE), 2012 IEEE Ninth International Conference on*, pages 327–332, Sept 2012.

[91] Ws-cdl — http://www.w3.org/tr/ws-cdl-10/.

[92] Wsmo — http://www.w3.org/submission/wsmo/.

[93] Aliaksei Yanchuk, Alexander Ivanyukovich, and Maurizio Marchese. A lightweight formal framework for service-oriented applications design. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 545–551. Springer Berlin Heidelberg, 2005.

[94] William Young and Nancy G. Leveson. An integrated approach to safety and security based on systems theory. *Commun. ACM*, 57(2):31–35, February 2014.

[95] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. *SAT*, 3, 2003.

[96] Haissam Ziade, Rafic Ayoubi, and Raoul Velazco. A survey on fault injection techniques, 2003.