RWTH Aachen

Department of Computer Science
Technical Report

Trustworthy Spacecraft Design Using Formal Methods

Viet Yen Nguyen

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2012-17

RWTH Aachen · Department of Computer Science · October 2013

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Trustworthy Spacecraft Design Using Formal Methods

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Viet Yen Nguyen, M.Sc.
aus
Hoorn, Nordholland, Die Niederlande

Berichter: Prof. Dr. Ir. Joost-Pieter Katoen
Prof. Dr. Paul Pettersson

Tag der mündlichen Prüfung: 4. Dezember 2012

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Viet Yen Nguyen
Lehrstuhl Informatik 2
`nguyen@cs.rwth-aachen.de`

Aachener Informatik Bericht AIB-2012-17

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

Model-based system-software co-engineering is a natural evolution towards meeting the high demands of upcoming deep-space and satellite constellation missions. It advocates better abstractions to cope with the increasing spacecraft complexity, and opens the door for a wide range of formal methods, benefiting from the mathematical rigour and precision they bring. This dissertation provides for both: we applied and evaluated state of the art modelling and analysis techniques based on formal methods to spacecraft engineering, and we developed novel theory that tackle issues encountered in industrial practice.

In particular, we formalised a modelling language by the aviation and automotive industry, namely the Architecture Analysis and Design Language (AADL). We show in this dissertation how we rooted it into the theories on discrete, real-timed and hybrid automata, various Markov models and temporal and probabilistic logics. This foundation enabled us to develop a comprehensive analysis toolset with model checkers being the cornerstone. It provides a wide range of analyses in an algorithmic fashion rather than the labour-intensive methods currently employed by the space industry. It can generate simulations, fault trees, failure modes and effects tables, performability curves, diagnosability artefacts and affirmations of correctness exhaustively. Our work has been subjected to extensive evaluation. At the European Space Agency, we applied it to a satellite design of one of the agency's ongoing missions. This dissertation reports on this case study. The case is currently the largest formal methods study of a satellite architecture reported in literature.

The sheer size and complexity of the satellite case study indicated several theoretical problems. To this end, we developed a reasoning theory based on Craig interpolants, that generates compositional abstractions from the model. It helps to understand large models, like the satellite case, more effectively. We furthermore studied the use of Krylov methods for improving the numerical stability of analysing a notorious class of Markov chains, namely, stiff Markov chains. They occur naturally in space systems where failure rates have large disparities. Our controlled experiments show that Krylov methods are superior in such cases.

Zusammenfassung

Modellbasiertes Co-Engineering von Systemsoftware stellt einen natürlichen Evolutionsschritt zur Erfüllung der hohen Anforderungen zukünftiger Weltraum- und Satellitenmissionen dar. Es bietet bessere Abstraktionsmöglichkeiten zum Umgang mit wachsender Komplexität von Raumfahrzeugen und ermöglicht den Einsatz einer breiten Auswahl an formalen Methoden, die sich durch ihre mathematische Stringenz und Genauigkeit auszeichnen. Die vorliegende Dissertation behandelt sowohl Grundlagen als auch Anwendungen: Wir demonstrieren und evaluieren den Einsatz modernster Modellierungs- und Analysetechniken basierend auf formalen Methoden für die Raumfahrt und entwickeln neue Theorien zum Umgang mit Problemen in einem industriellen Umfeld.

Konkret wird eine in der Luft-, Raumfahrt- und Automobilindustrie verbreitete Modellierungssprache namens “Architecture Analysis and Design Language” (AADL) formalisiert. Wir stellen ihre Verwurzelung in den Theorien der diskreten, Echtzeit- und Hybridautomaten, verschiedenen Markov-Modellen, sowie temporaler und probabilistischer Logik vor. Diese Grundlagen ermöglichen uns die Entwicklung eines umfangreichen Analysewerkzeugs basierend auf Modelcheckern. Es bietet eine breite Auswahl an algorithmischen Analysen anstatt der aufwändigen manuellen Methoden, welche zurzeit in der Raumfahrtindustrie eingesetzt werden. Dazu unterstützt es die vollautomatische Generierung und Analyse von Systemsimulationen, Fehlerbäumen, “failure modes and effects”-Tabellen, Wahrscheinlichkeitskurven, Diagnoseartefakten und Korrektheitsberprüfungen. Die Methoden werden durch ausführliche Evaluierungen validiert. Bei der Europäischen Raumfahrtagentur (ESA) wurden unsere Techniken während der Entwicklung einer zukünftigen Satellitenmission angewendet, deren Ergebnisse in der vorliegenden Dissertation behandelt werden. Diese Fallstudie ist die größte, in der Literatur erwähnte Studie zum Einsatz formaler Methoden zur Modellierung und Analyse einer Satellitenarchitektur.

Die schiere Größe und Komplexität dieser Fallstudie stellte uns vor einige Probleme theoretischer Natur. Hierzu entwickelten wir Theorien zur Schlussfolgerung, basierend auf Craig-Interpolationen, die kompositionelle Abstraktionen des Modells generieren. Diese unterstützen das Verständnis großer Systemmodelle. Des Weiteren untersuchen wir die Verwendung von Krylov-Methoden zur Verbesserung der numerischen Stabilität bei der Analyse einer spezieller, sogenannter “steifer”

Markov-Ketten. Diese treten häufig in Raumfahrtsystemen auf, bei denen die Ausfallraten von Komponenten große Diskrepanzen aufweisen. Unsere Experimente zeigen, dass Krylov-Methoden in diesen Fällen überlegen sind.

(Translation from the English abstract by Tim Lange and Thomas Noll)

Management Summary

This dissertation describes novel software tools, techniques and theories from the science of formal methods to improve state-of-the-art spacecraft engineering. Formal methods researchers and practitioners emphasise the expression of hardware and software systems using a formal system (i.e. logical system), benefiting from its mathematical properties, as consistency, validity, soundness and completeness. These qualities provide clarity and unambiguity to the engineers' understanding of the spacecraft in development, despite of its many interrelations and complexities. The latter two are rising steadily due to the trends of increasing mission demands regarding safety and dependability as well as our increasing technological capability. The effect is particularly visible during assembly, integration and testing. Increasingly more issues, in particular involving the fault management systems, are encountered at this phase. Resolving them with the desired level of effectiveness is typically sustained by increasing costs and staffing, as an increase of resolving time is often infeasible due to the strict launch windows. Using formal methods, the spacecraft is modelled early in its engineering lifecycle (e.g. phase 0 to phase C), which enables a wide-range of automated formal analyses performing early verification and validation on the current design. Many issues currently encountered during assembly, integration and testing, are thus resolved earlier.

Our modelling formalism is christened SLIM, the System-Level Integrated Modelling language. It is a formal dialect of AADL, the Architecture, Analysis and Design Language. It provides a hierarchical and component-oriented language to express system, hardware, software and erroneous aspects in a coherent manner. System, hardware and software components interact through event and data ports, and their nominal behaviour is expressed using a (hybrid) state transition system. Erroneous behaviour is expressed as a probabilistic state machine. A mechanism called model extension is provided to automatically combine the nominal components and the erroneous components into an extended system model. The latter describes a single integrated model covering the system's nominal, erroneous and degraded behaviour.

Upon SLIM, we mapped a wide-range of verification and validation methods. These methods are automated forms of manual methods of analysis typically performed in a spacecraft engineering lifecycle. Functional correctness analysis checks whether a design meets its functional requirements. It does this by employ-

ing model checking, which is an exhaustive technique of verifying a requirement against all possible system traces. Safety and dependability analyses are provided through the generation of dynamic fault trees and failure modes and effects tables (FMEA) from the SLIM model. Risks can be probabilistically assessed by transforming the fault tree to its underlying continuous-time Markov chain. For quantifying the system performance under degraded modes, performability analysis provides for cumulative distribution functions plotted as graphs. For analysing the effectiveness of fault management systems specifically, we provide for failure detection, fault isolation and failure recovery analysis. These respectively analyse which observables are triggered upon occurrence of faults, which combination of faults trigger observables, and determine whether faults are recoverable. Lastly, we provide for diagnosability analysis. Diagnosability analyses can prove whether the observables provide sufficient information to correctly infer a fault. All these automated analyses can be employed to determine whether the current design, or competing designs, meet functional, safety, dependability and performance requirements. They can also be used to determine whether a design is over-engineered, and thus provide indication for unnecessary system complexity.

The whole approach is software tool-supported and can be run on ordinary PC workstations. A graphical drag-and-drop interface called the COMPASS Graphical Modeller can be used to create a SLIM model. The resulting model can be loaded into the COMPASS toolset, which is an engineer-friendly graphical environment for running analyses and investigating their outputs. The technological readiness level of the tools is laboratory-tested.

As part of this dissertation, an extensive case study was conducted by developing the largest and most comprehensive system-level model of a spacecraft so far. Using the deliverables of an ESA satellite in development, its full platform has been modelled and analysed. This was done in parallel with the actual development, this to avoid the bias of modelling a completed, matured and issue-free design after development. The resulting model has a nominal state space over 48 million states, and when fault injections are injected the state space grows up to $2^{18} \cdot$ 48 million states. Its large size led us to hit practical computation limits of the algorithms underlying performability and diagnosability analysis, hinting directions for future theoretical research. Furthermore, significant practical experience of using formal methods in an engineering process was gained. It showed the need for conscious documentation and characterisation of abstraction levels and maintenance of traceability between the SLIM model and the spacecraft deliverables. The resulting model serves as a guiding reference for future formal modelling initiatives.

The case study clearly indicated the need for handling the size and complexity of spacecraft models by the model checking algorithms underlying the formal analyses. As a response, we developed a novel theory to exploit the compositional structure of compositional modelling languages by synthesizing a small environment of a component in isolation. The synthesis is achieved through a logical interpolation of the component's environment transitions. They are reduced to the variables which are only used for interaction with the component, thus abstracting indirectly (ir)relevant system variables. The environment provides insight on the possible interactions of the component, thus clearly characterising possible system-software boundaries, and/or system-fault management boundaries.

We also investigated the use of Krylov subspace methods for computing the transient probabilities of continuous-time Markov chains. Transient probability computation is the underlying analysis for probabilistic risk assessment. We observed using a controlled experiment that Krylov subspace methods performs better on stiff Markov chains. These Markov chains arise typically from systems that have large disparities between the component failure rates.

Conclusive, we developed a state-of-the-art modelling formalism (SLIM) and a toolset (COMPASS) to support early verification and validation of functional, safety, dependability and performance aspects in the spacecraft engineering lifecycle. The modelling formalism and toolset have been extensively evaluated, and our own satellite platform case study being the most extensive one. The sheer size of the latter case study inspired to advance the algorithms underlying the formal analyses. We thus developed a compositional reasoning theory that copes better with the increasing size and complexity of future spacecraft. We furthermore report on the use of Krylov subspace methods, and demonstrated their superiority of more traditional and commonly-used methods. The overall result of is a set of novel software tools, techniques and theories that are better equipped for dealing with the increasing size and complexity of future spacecraft.

Acknowledgements

This thesis describes the results of cumulative group-efforts conducted by many people, that include myself and my respected colleagues. For this as the result, I have many people to thank for.

A great part of my gratitude goes towards my professor, Joost-Pieter Katoen, who provided me many opportunities and enduring freedom for my research as well as advice and support when I needed it. The same gratitude also goes to Thomas Noll, with whom I closely cooperated throughout the years, and who always made time available for advice and support. A significant part of this thesis bears his contribution.

During the first years, we embarked on an intense cooperation with Alessandro Cimatti, Marco Bozzano, Marco Roveri and Roberto Cavada, during which we did not only shared the heavy workload, but also the resulting successes with much content. Their contributions are also interwoven in this thesis. Afterwards, we also had the pleasure to work with Pierre Dissaux, Jerome Legrand and Arnaud Schach. Your collegiality has been much enjoyed.

Much of the work on COMPASS and my teaching activities were also made possible by my assistants, of which a select few worked with me for several years. They are, ordered by their seniority of service, Christian Dehnert, Benedikt Brütsch, Falko Dulat, Friedrich Gretz, Gereon Kremer, Christina Jansen, David Piegdon, Chitra Hapsari Ayuningtyas, Benjamin Bittner and Clarence Xu. I thank you all for your efforts and contributions.

My stays at the technical heart of the European space sector, ESTEC, were a great source of inspiration and joy. I am foremost thankful for Yuri Yushtein's support and efforts that made the stays possible. My thanks also go to Marie-Aude Esteve, Jean-Loup Terraillon and Konstantinos Mokos for their valuable input and perspectives and Xavier Olive for securing funds for my research.

Finally, I thank Maximilian Odenbrett, Bernhard Ern and Bart Postma, who earned their Masters degree under my supervision. Their fresh perspectives have led to new research and publications, for which I am pleased that I was part of that.

Viet Yen Nguyen

Contents

1 Introduction 1
1.1 Going Into Space 1
1.2 Unambiguity is Understanding 2
1.3 Contributions 3
1.4 Outline 5
1.5 Publications 6

2 System-Software Co-Engineering for Space 8
2.1 Systems Engineering 8
2.2 Software Engineering 17
2.3 Fault Management Engineering 23

3 Formal Architecture Modelling 28
3.1 Nominal Behaviour 28
3.2 Erroneous Behaviour 33
3.3 Model Extension 35
3.4 Formal Semantics 39
3.5 Graphical Notation 50
3.6 Differences between SLIM, AADL and Annexes 51
3.7 Discussion 54

4 Formal Architecture Analysis 57
4.1 Fault Injection 57
4.2 Properties 60
4.3 Simulation 64
4.4 Model Checking 64
4.5 Fault Tree Generation 70
4.6 Probabilistic Fault Tree Evaluation 73
4.7 Probabilistic Fault Tree Verification 77
4.8 Failures, Modes and Effects Table Generation 78
4.9 Fault Tolerance Evaluation 79
4.10 Diagnosability 81

4.11 Fault Management Effectiveness 84
4.12 Performability Analysis 87

5 Inside the COMPASS Toolset 93
5.1 Building Blocks 93
5.2 Extensions 95
5.3 Console Interface 97
5.4 Graphical Interface 97
5.5 COMPASS Graphical Modeller 98

6 Satellite Platform Case Study 100
6.1 Case 100
6.2 Objectives 102
6.3 Modelling 103
6.4 Requirements Specification 106
6.5 Analyses 107
6.6 Discussion 110

7 Craig Interpolation-Based Compositional Reasoning 115
7.1 Preliminaries 115
7.2 Component-Oriented Interpolation 121
7.3 Applications 125
7.4 Experimental Evaluation 126
7.5 Discussion 132

8 Krylov-Based Transient Analysis of CTMCs 135
8.1 Stiffness 136
8.2 Model Checking Markov Chains 136
8.3 Krylov Subspace Methods 137
8.4 Experiments 142
8.5 Discussion 149

9 Conclusion 153

A Sensor-Filter Model 156
A.1 Nominal Model 156
A.2 Error Model 160

Standards, Handbooks & Manuals 162

Publications 165

1

Introduction

Space flight is one of the greatest achievements in modern times, utilising the full body of formal and natural sciences developed so far. And now, the continuing human presence in low Earth orbit through the International Space Station sets a distinct mark on our technological capabilities. The ambition is now set to the Moon, Mars and beyond, with exploration programs and spacecraft already present and newer ones on course. Going deeper into space presents tougher challenges and as such, requires a mastery of technology that has yet to be developed. In line with this, the European Space Agency (ESA) and Thales Alenia Space funded the research resulting in this dissertation, which specifically aims to improve the system software of spacecrafts using state of the art formal methods, benefiting from its mathematical rigour and preciseness.

In this chapter, the prime technological aspect of space-flight is briefly introduced, namely the spacecraft, and the engineering challenges it presents. It is followed by an introduction to formal methods, and concluded with an overview of industrial and academic contributions described in this dissertation.

1.1 Going Into Space

The spacecraft is the vehicle used for travel in space, and together with the launcher (i.e. rocket) and ground systems is one of the main ingredients for a typical mission. Spacecraft are typically tailored to the mission and its constraints. For example, a spacecraft that has to operate on Mars needs to account for longer communication latencies than spacecraft that orbit Earth. Scientific and Earth-observation spacecraft on the other hand typically have high bandwidth demands, and this has consequences on the power requirements. Tailored systems like this require sizeable investments. These investments are of such a degree that a spacecraft's failure to meet mission objectives makes it worthwhile to ensure the first mission attempt is immediately successful. Through decades of experience with space flight, a development process for spacecraft emerged that led to safe and dependable systems. In the European tradition, the main activities are mission analysis, requirements analysis, design definition, verification, production, utilisation and disposal. To control the process, these activities are part of mission phases, during which a strict schedule of review meetings are planned at which ESA specialists review all artefacts developed by the contracting company. At the reviews, the coherence, completeness and consistency of the artefacts are checked for, and if they meet mission phase requirements, ESA's review board gives approval to enter the subsequent phase. Artefacts range from technical documents containing e.g. design schematics, test reports, justification reports, models, source code to more physical evidences like early flight prototypes and the eventual system itself. One trend in the space industry is that the increasing demands on the mission have led to an increasing prominence of software in the system. Software provides unprecedented functionality to meet mission requirements like for example increasingly more accurate control algorithms to keep stricter orbits and flight-plans, to highly automated self-healing systems that enable unprecedented safety and reliability. It is for this, software has become an important factor for the mission's realisation and its success. Its prominence is to such a degree that software impacts the whole system in many interweaving ways. Some colloquially say that a modern spacecraft is a flying computer, which is a strong contrast with spacecraft from the sixties, where nearly all computations were performed on analog signals. The provision of sufficient artefactual evidence of the coherence, completeness and consistency of software to ESA's review board requires a thorough understanding of its relation and interaction with the overall system. Due to the enormous possibilities of software, and its near-omnipresence in spacecraft, gaining such an understanding has become increasingly challenging. We argue that formal methods is the answer to meet this challenge. It brings the clarity and rigour from mathematical logic to the system software domain, providing the means to effectively develop systems of unprecedented functionality and enabling missions which were previously difficult to realise.

1.2 Unambiguity is Understanding

Formal methods are a particular kind of techniques that emphasize expression of hardware and software systems using a formal system (i.e. logical system). A typical formal system has favourable properties as consistency (i.e. no arguments contradict to each other), validity (i.e. an argument's conclusion is entailed by its premises), soundness (i.e. only valid arguments with true premises are used) and completeness (i.e. all true conclusions are derivable from the axioms through arguments). These qualities enable an unambiguous expression of the system under development and as such its interpretation is the same among different persons. This clarity of understanding aids in the development of large and complex interacting systems, where it is human to lose perspective and sight in all interrelations.

Formality can be exercised in many ways. Natural language in the form of prose for example, obeys the rules of a language's syntax and grammar, yet embeds a great deal of semantics within the words, allowing for varying interpretations. Structured prose typically restricts points of unambiguity, and thus is considered to be more formal. Logics and automata are typically considered as the most formal of formal methods for expressing systems. These expressions can be used in various points in a system's development process. A formal model can be used to express a system and a formal logic can be used to express its requirements. Thus, it can clarify the system's specification. If the used formal system is sufficiently formal, one can derive (i.e. prove) properties from formal models and logics using the rules of the formal system. This aids the verification and validation process, during which desirable system's properties are checked for. At the highest level of formality, automation of proofs becomes possible if the formal system satisfies particular conditions, e.g. finiteness. The automation can be provided in the form of software tools implementing algorithms for proving, avoiding human errors in the construction of proofs and enabling analysis of systems with larger size and detail due to the increasing computing performance. This dissertation aligns with that particular class of formal methods. We argue that the space industry benefits most from formal methods if it is elevated to the degree of automated analysis.

1.3 Contributions

This dissertation is a mixture of theory applied to practice and practice inspired theory. It was able to form through the unique opportunities we enjoyed. We, together with Fondazione Bruno Kessler, were commissioned by ESA to develop a comprehensive toolset for modelling and analysis of spacecraft using state-of-the-art-formal methods for improving system-software co-engineering. Through its industrial evaluation by Thales Alenia Space, which participated by running case studies and providing their industrial engineering experience, we gained a deeper understanding of best practices for applying our theories to industrial practice. This was the Correctness, Modelling and Performance of Aerospace Systems (COMPASS) project and it ran from 2008 to 2010. A small extension project was commissioned to us and Ellidiss in 2011 to develop a graphical modelling environment for COMPASS's modelling language. From 2010 onwards, we received research grants by ESA and Thales Alenia Space for the reverse: develop theories to tackle the problems in practice. Part of this was an one year (cumulatively) visit to the European Space Research and Technology Centre (ESTEC), which is the technical heart of the European space industry employing over 2500 persons. We were provided resources that are typically out of reach for academics, like an in-house training in space system engineering and its verification and validation, access to technical designs of spacecraft in development, allowing us hands-on experience with the domain of discourse and the possibility of inquiry to ESA specialists, learning from their skill, knowledge and experience. All these were a great source of inspiration for our developed theory.

As this dissertation flows from theory and practice, it is only natural that it contributes to both sides. Towards industry, its contributions are:

- An AADL-like (cf. Chapter 3) formal modelling language, christened SLIM, enabling engineers to express both nominal, erroneous, and their interweaving behaviour in a formal, yet user-friendly way.
- A mapping of correctness, performance, safety and dependability analyses to proofs by (probabilistic) model checking, enabling engineers formal analyses of SLIM models.
- The COMPASS toolset, implementing the aforementioned formal analyses as automated programs accessible through a graphical interface.
- Industrial case studies, demonstrating benefits and limitations of state-of-the-art formal modelling and analysis.
- A formal reference model of a satellite platform, enabling engineers to quickly pick up formal modelling.
- A set of modelling guidelines, capturing best practices that avoid typical pitfalls and leverage existing experience.
- Optimisation to functional correctness verification and validation by compositional reasoning, enabling models of higher complexity and fidelity to be analysed.
- Introduction of a highly numerical stable algorithm for probabilistic risk assessment, enabling assessments of systems with large disparities in failure rates.

From an academic perspective, this dissertation's contributions are the following:

- A formal semantics for a component-oriented language that supports dynamic reconfiguration and covers action, timed, hybrid and probabilistic aspects.
- A notion of model extension that interrelates functional behaviour with erroneous behaviour through fault injections using data-failures.
- An approach towards performability by interpreting extended models using the formal semantics as interactive Markov chains.
- An algorithm for compositional abstraction of a component's environment using Craig interpolation.
- An evaluation of Krylov subspace methods and the observation that they perform well on stiff continuous-time Markov chains, along with an explanation for this performance.

- A large-scale industrial case study as a reference model for comparing and evaluating algorithms for formal analysis.

We also contributed to the development of a notion of slicing over SLIM models [ONN10] and a notion of impact isolation over SLIM models [Ern12]. These are however not part of this dissertation.

1.4 Outline

- In Chapter 2 we discuss the system and software engineering life-cycles and focus on their verification and validation activities. A section is devoted to a special case of system software engineering, namely fault management engineering.
- In Chapter 3 we introduce our component-oriented modelling language called SLIM, along with our approach towards model extension and a definition of its formal semantics in terms of a network of event-data automata.
- In Chapter 4 we describe methods for formal analysis over SLIM models. Simulation, model checking, fault tree generation/evaluation/verification, FMEA table generation, fault tolerance evaluation, diagnosability, fault detection/isolation/recovery and performability are discussed.
- In Chapter 5 we provide an overview of the inner-workings of the COMPASS toolset. It describes which software components were reused, adapted and freshly developed.
- In Chapter 6 we introduce our large scale case study of a satellite platform of an ongoing ESA project, our verification and validation results and experiences gained from creating it.
- In Chapter 7 we describe our novel approach for compositional reasoning and invariant verification of compositional modelling languages using Craig interpolation and bounded model checking.
- In Chapter 8 we report on our investigation of using Krylov subspace methods for computing transient probabilities over continuous-time Markov chains. We observed that these methods have a better performance on stiff Markov chains, and that this can explained by an analysis of the eigenvalues of the matrix representing the Markov chain.
- In Chapter 9 we reflect back on our work and describe future research lines of industrial interest.

1.5 Publications

The origins of most chapters are from the COMPASS project deliverables (not publicly available) and our publications, which are the following

- M. Bozzano, A. Cimatti, M. Roveri, J.-P. Katoen, V.Y. Nguyen and T. Noll. 'Codesign of Dependable Systems: a Component-Based Modeling Language'. In: *Proceedings of the 7th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2009, pp. 121–130.

- M. Bozzano, A. Cimatti, M. Roveri, J.-P. Katoen, V.Y. Nguyen and T. Noll. 'Verification and Performance Evaluation of AADL models'. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Ed. by H. van Vliet and V. Issarny. ACM, 2009, pp. 285–286.

- M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll and M. Roveri. 'The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems'. In: *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*. Ed. by B. Buth, G. Rabe and T. Seyfarth. Vol. 5775. Lecture Notes in Computer Science. Springer, 2009, pp. 173–186.

- M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll and M. Roveri. 'Model-Based Codesign of Critical Embedded Systems'. In: *Proceedings of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*. Ed. by S. van Baelen, T. Weigert, I. Ober and H. Espinoza. Vol. 507. CEUR, 2009, pp. 87–91.

- M. Bozzano, R. Cavada, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll and X. Olive. 'Formal Verification and Validation of AADL Models'. In: *Proceedings of the 4th Conference on Embedded Real Time Software and Systems Conference (ERTS^2)*. AAAF & SEE. 2010.

- M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, M. Roveri and R. Wimmer. 'A Model Checker for AADL'. In: *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*. Ed. by T. Touili, B. Cook and P. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 562–565.

- F. Dulat, J.-P. Katoen and V.Y. Nguyen. 'Model Checking Markov Chains using Krylov Subspace Methods: An Experience Report'. In: *Proceedings of the 7th European Performance Engineering Workshop (EPEW)*. Ed. by A. Aldini, M. Bernardo, L. Bononi and V. Cortellessa. Vol. 6977. Lecture Notes in Computer Science. Springer, 2010, pp. 115–130.

- M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll and M. Roveri. 'Safety, Dependability and Performance Analysis of Extended AADL Models'. In: *Computer Journal* 54.5 (2011), pp. 754–775.

- Y. Yushtein, M. Bozzano, A. Cimatti, J. Katoen, V.Y. Nguyen, T. Noll, X. Olive and M. Roveri. 'System-Software Co-Engineering: Dependability and Safety Perspective'. In: *Proceedings of the 4th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. IEEE, 2011, pp. 18–25.

- M.-A. Esteve, J.-P. Katoen, V.Y. Nguyen, B. Postma and Y. Yushtein. 'Formal Correctness, Safety, Dependability and Performance Analysis of a Satellite'. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012.

An exception to this is Chapter 7, which is a (yet) unpublished result. Publications published during the COMPASS project and afterwards, but that are not discussed in this dissertation are the following

- V.Y. Nguyen and T.C. Ruys. 'Incremental Hashing for SPIN'. In: *Proceedings of the 15th International Conference on Model Checking Software (SPIN)*. Ed. by K. Havelund, R. Majumdar and J. Palsberg. Vol. 1556. Lecture Notes in Computer Science. Springer, 2008, pp. 232–249.

- V.Y. Nguyen and T.C. Ruys. 'Memoised Garbage Collection for Software Model Checking'. In: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by S. Kowalewski and A. Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 201–214.

- N.H. Aan De Brugh, V.Y. Nguyen and T.C. Ruys. 'MoonWalker: Verification of .NET Programs'. In: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by S. Kowalewski and A. Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 170–173.

- M.R. Odenbrett, V.Y. Nguyen and T. Noll. 'Slicing AADL Specifications for Model Checking'. In: *Proceedings of the 2nd NASA Formal Methods Symposium (NFM)*. Ed. by C. Muñoz. NASA Conference Proceedings, 2010, pp. 217–221.

- V.Y. Nguyen and T.C. Ruys. 'Selected Dynamic Issues in Software Model Checking'. In: *Software Tools for Technology Transfer (STTT)* 15.4 (2013), pp. 337–362.

2

System-Software Co-Engineering for Space

The act of verification and validation is about improving the understanding of a system (under development) in order to make informed decisions. Unknown or vaguely understood behaviours lead to risky decisions, whereas known and understood behaviours typically lead to trustworthy decisions. The degree of verification and validation hence depends on the amount of risk one is willing to accept. In the European tradition of space engineering, a thorough verification and validation process is applied throughout the overall system engineering life-cycle, that prioritises investigation and understanding of safety-critical behaviours. A strongly increasing contributing factor to these behaviours is software. Software is present in nearly all parts of the system in intertwining and apparent peculiar ways. Furthermore, software is flexible for modification and adaptation, yet, changing it might require a re-investigation of the system's possible (new) behaviours. Thus, it has become increasingly challenging to effectively verify and validate software (under development), and especially its interaction with the rest of the system.

This chapter overviews the European space industry practices on system engineering, software engineering and their verification and validation. In the end of this chapter, a section is dedicated to fault management engineering (also known as FDIR: fault/failure detection, isolation and recovery), which is a special case of system-software engineering. We argue that it poses the greatest challenge in engineering space systems.

2.1 Systems Engineering

The technological and operational part of a space mission is the *space system*. In the following, a typical space system is outlined, along with its life-cycle and the applicable standards. The contents here reflects practices codified in the ECSS

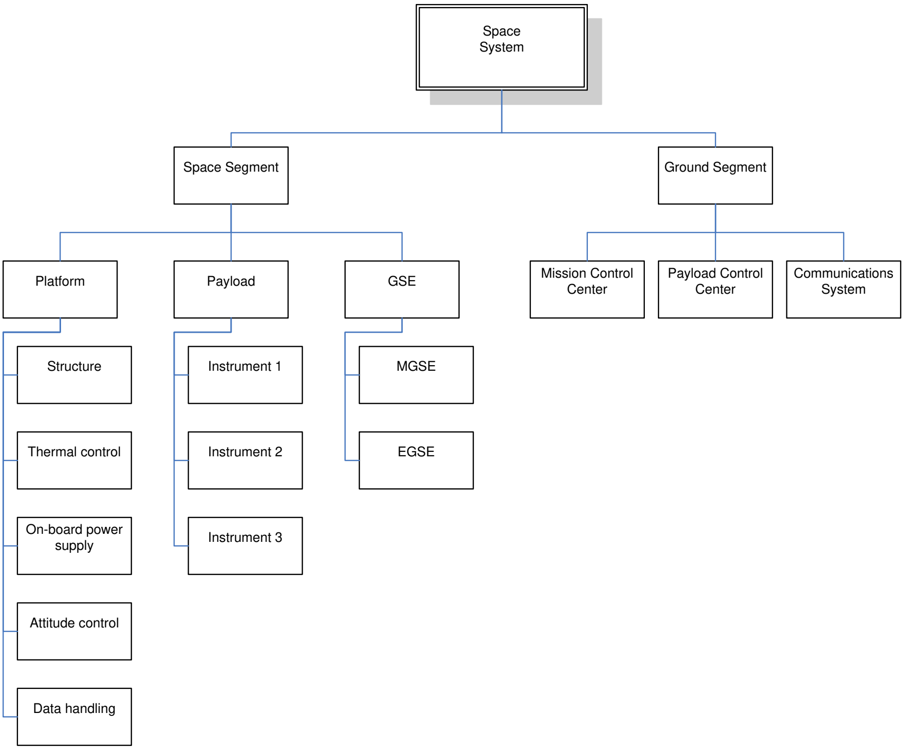

Space System
Space Segment
Ground Segment
Platform
Payload
GSE
Mission Control Center
Payload Control Center
Communications System
Structure
Thermal control
On-board power supply
Attitude control
Data handling
Instrument 1
Instrument 2
Instrument 3
MGSE
EGSE

Figure 2.1: An example (incomplete) breakdown of a space system. Source: [ECSS-M-ST-10C].

standards (European Cooperation for Space Standardisation). These standards are governed by the European Space Agency.

2.1.1 Product Tree

The space system (see Figure 2.1) is generally composed of two segments: the space segment and the ground segment.

The space segment typically consists of a spacecraft, and in some cases, multiple spacecraft (e.g. a satellite communication network). Spacecraft are typically further composed of a platform and a payload part. The payload contains mission-specific instruments and equipment (e.g. a communication repeater, earth observation sensory, telescopes, navigation signal emitter), whereas the platform keeps the spacecraft in space and provides proper conditions for the payload to function. The ground support equipment (GSE) is used to service spacecraft while still on the ground. They are typically electric (EGSE) or mechanical (MGSE) in nature. The platform is further decomposed in subsystems (e.g. structure, thermal

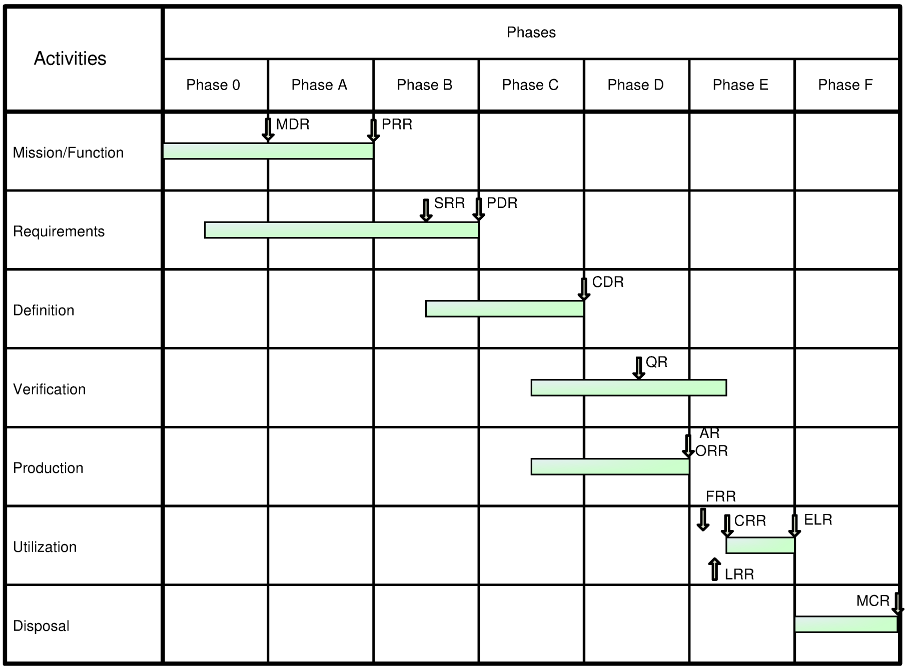

Activities
Phases
Phase 0
Phase A
Phase B
Phase C
Phase D
Phase E
Phase F
Mission/Function
Requirements
Definition
Verification
Production
Utilization
Disposal
MDR
PRR
SRR
PDR
CDR
QR
AR
ORR
FRR
CRR
ELR
LRR
MCR

Figure 2.2: ESA System engineering life-cycle. Source: [ECSS-M-ST-10C].

control, power supply, attitude control, data handling).

The ground segment typically consists of control centres, along with the communications systems. The control centres operate the space segment in order to fulfil the mission. Since the space segment is not always directly within communication range from a single location on Earth (e.g. on the other side of the Earth), the larger space faring nations maintain an extensive communications network across the globe which relays commands from a central control centre.

Within the scope of this dissertation, the spacecraft in the space segment are typically of greater interest. The technical requirements for spacecraft have demanding requirements as they have to function in a harsher and more alien environment. Additionally, they cannot be serviced/repaired easily by physical human intervention.

2.1.2 Systems Engineering Life-Cycle

The typical life-cycle of a space system is depicted in Figure 2.2. It shows that the life-cycle starts in phase 0, which is mission analysis. During this phase, needs are analysed and a mission outline is characterised, along with its expected performance and the required technological resources. The end of phase 0 is marked by a mission definition review (MDR). Afterwards, during phase A, the feasibility is analysed. Management plans, engineering plans and assurance plans are set up, and the functional aspects are characterised. Critical technologies are identified. System and operation concepts are crafted, including a design philosophy and a verification approach. The programmatic risks are also charted. The end of phase A is marked by a preliminary requirements review (PRR). In phase B, all plans and schedules are finalised. Trade-off studies are conducted to determine the preferred system and operation concepts. Afterwards a preliminary design of the system is defined, along with a refinement of its requirements. The initial reliability and safety levels are assessed, and the risk assessments are updated. Two milestones are present in phase B, namely the systems requirements review (SRR) and the preliminary design review (PDR). When this point is reached, the system requirements and functions are elaborated, technical and programmatic constraints are identified, activities and resources are properly planned and scheduled, and the risks are assessed. The results described in this dissertation apply in particular to phase B.

For the sake of completeness, the remaining phases shall be shortly described. In phase C, the detailed definition phase, and phase D, qualification phase, comprise the development and qualification activities. The milestones in these phases are the critical design review (CDR), qualification review (QR), acceptance review (AR) and operation readiness review (ORR). Phase E, the operations and utilisation phase, is generally the most cost-expensive phase. It comprises of space launch, commission and utilisation of the space system. The milestones are the flight readiness review (FRR), launch readiness review (LRR), commissioning result review (CRR) and end-of-life review (ELR). In phase F, the disposal phase, the space system is safely disposed and a mission close-out review (MCR) is held.

This phased approach along with the review meetings are present to control change, risk and work division from a programmatic perspective. In the next subsection, it will be described how verification and validation is used to control technical change, risk and work division, and how this impacts the system engineering life-cycle.

2.1.3 Technical Risks

Spacecraft face specific hazards, for which no consideration is needed if such a system would only be used on Earth. The hazards, and their associated risks, depend heavily on their mission profile, and thus between space missions, hazards may vary widely. In this subsection, the hazards are summarised, providing a justification for the commonly used verification and validation methods that are described later on.

A spacecraft needs to be launched with a launch vehicle (i.e. rocket). During the first minutes of launch, the spacecraft is under enormous mechanical stress. Vibrations, shocks, heat and the transition to vacuum are typical for all launchers, and need to be coped with. Once the launcher reaches the aimed orbit, the spacecraft is separated from the launcher using a separation mechanism, which needs to operate while being sufficiently rigid to handle the launch hazards.

Deeper in space, the Earth's gravitational pull weakens and other gravitational forces may apply (e.g. by the Sun, Mars, Moon). These conditions are difficult to mimic on a grand-scale on Earth. Mechanical system behaviour needs to be designed appropriately. In low Earth orbits (LEO), spacecraft may experience atmospheric drag, which affects attitude and could shorten orbital lifetime. This may need countermeasures. Then there are thermal issues. Spacecraft in general operate in the coldness of space, and appropriate measures need to be taken to handle that. Spacecraft that go close to the Sun on the other hand experience intense heat on their Sun-facing side, whereas the other side might be extremely cold. Additionally, spacecraft might experience issues due to radiation and charged particles (e.g. plasma effects). Such issues do not emerge on Earth due to the presence of a strong magnetosphere.

Another class of issues are due to the sheer size of space, and that the laws of physics apply on a larger scale. On deep space missions, the long distance between Earth and the spacecraft impair constraints on communications. It might takes many minutes to have a signal reach a spacecraft on Mars up to hours to reach a spacecraft near Pluto. Bandwidth might also be an issue. Missions requiring lots of bandwidth (e.g. telescopes) could use high-frequency bandwidths, but this increases power usage of the radio. These issues are not of much concern on Earth due to its small size and its abundance of power generation facilities.

2.1.4 System Verification & Validation

The terms verification and validation are often used in one breath, indicating the need for increased understanding of the system (under design), and thereby decreasing programmatic risk by knowing the technical risk. Different engineering branches throughout different industries have developed varying interpretations and definitions for these terms. Within the European space industry, the following definitions from [ECSS-P-001B] are used:

- Verification is the "confirmation through the provision of objective evidence that specified requirements have been fulfilled."
- Validation is the "confirmation through the provision of objective evidence that the requirements for a specific intended use or application have been fulfilled."

Essentially, these boil down to respectively "did we build the system right?" and "did we build the right system?". The distinction is important, because verification emphasises correctness with respect to the requirements, whereas validation emphasises correctness with respect to the stakeholders intents. Both are needed. Requirements could be incorrect, and stakeholders intents cannot be checked at each phase in the development process. Note that the stakeholder(s) could be different persons in the process. For example, for ground control software, a software engineer can consider the satellite operator as the stakeholder, whereas the system engineer might consider the mission program manager as the stakeholder. These perspectives impact the choice of verification and validation methods.

Planning

Figure 2.2 might suggest that verification (and validation) activities are performed during production and especially afterwards, to assess the quality of the system implementation. This is its traditional scope. For critical systems however (e.g. space systems), the risks are too high for a single verification and validation cycle. That is why verification and validation activities are performed throughout its whole life-cycle (see Figure 2.3).

The verification and validation activities in each phase differ depending on the foreseen available system artefacts. During phase A for example, emphasis is put on requirements validation, whereas in phase B, emphasis is put on model validation. During phase D however, emphasis could be put on integrated verification. Also the granularity of activities can vary. For example, a verification and validation campaign for a processor module differs much from that of a thermal subsystem. All activities are tailored based on needs and expectations.

The outputs of verification and validation activities are usually reports, verification matrices and traceability tables. The latter two are used to verify completeness of the verification and validation activities. The outputs are used to refine existing plans, like the schedule and allocated resources (e.g. manpower, test-facilities) for upcoming verification and validation activities due to the increased understanding of the system. It is also used to revise the system design, the requirements or the mission profile, depending on whether that aligns with the stakeholders intents. In the end, these activities aim to increase understanding of the system and basing informed decisions on that knowledge.

Within this dissertation, the early verification and validation activities, i.e. those during phase A and B, are of particular interest. Decisions made during these phases have great impact on the subsequent phases. Using our formal modelling and analysis techniques, we argue that an increased understanding of the system can be gained upon the existing approach of verification and validation.

Methods

Verification and validation activities are performed on system artefacts from which statements about the eventual flown system can be derived. Roughly speaking, we distinguish three categories of system artefacts, namely documents, models and implementations. Documents are printable artefacts that describe (usually in prose-form) a particular aspect of the system. It could for example describe requirements, justifications, utilities, trade-off, sizing, feasibility analyses, compli-

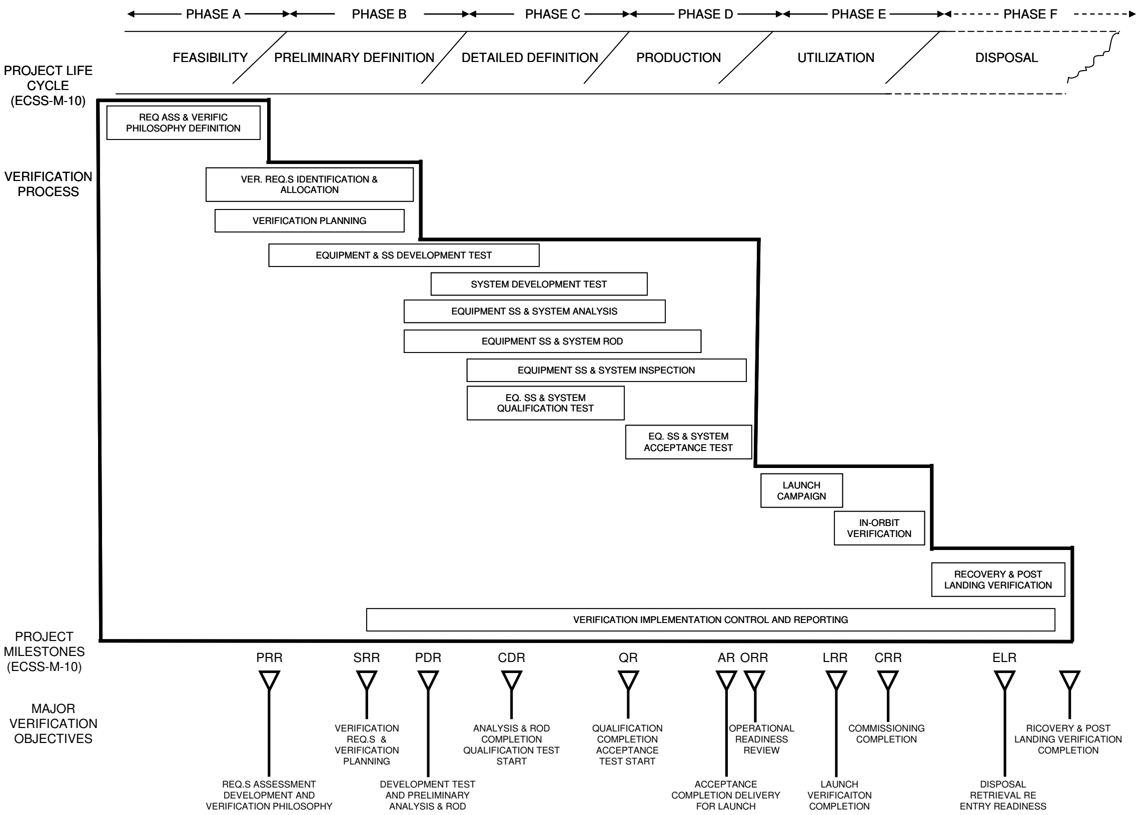

PHASE A
PHASE B
PHASE C
PHASE D
PHASE E
PHASE F
PROJECT LIFE CYCLE (ECSS-M-10)
FEASIBILITY
PRELIMINARY DEFINITION
DETAILED DEFINITION
PRODUCTION
UTILIZATION
DISPOSAL
VERIFICATION PROCESS
REQ ASS & VERIFIC PHILOSOPHY DEFINITION
VER. REQ.S IDENTIFICATION & ALLOCATION
VERIFICATION PLANNING
EQUIPMENT & SS DEVELOPMENT TEST
SYSTEM DEVELOPMENT TEST
EQUIPMENT SS & SYSTEM ANALYSIS
EQUIPMENT SS & SYSTEM ROD
EQUIPMENT SS & SYSTEM INSPECTION
EQ. SS & SYSTEM QUALIFICATION TEST
EQ. SS & SYSTEM ACCEPTANCE TEST
LAUNCH CAMPAIGN
IN-ORBIT VERIFICATION
RECOVERY & POST LANDING VERIFICATION
VERIFICATION IMPLEMENTATION CONTROL AND REPORTING
PROJECT MILESTONES (ECSS-M-10)
PRR
SRR
PDR
CDR
QR
AR
ORR
LRR
CRR
ELR
MAJOR VERIFICATION OBJECTIVES
VERIFICATION REQ.S & VERIFICATION PLANNING
ANALYSIS & ROD COMPLETION QUALIFICATION TEST START
QUALIFICATION COMPLETION ACCEPTANCE TEST START
OPERATIONAL READINESS REVIEW
COMMISSIONING COMPLETION
RICOVERY & POST LANDING VERIFICATION COMPLETION
REQ.S ASSESSMENT DEVELOPMENT AND VERIFICATION PHILOSOPHY
DEVELOPMENT TEST AND PRELIMINARY ANALYSIS & ROD
ACCEPTANCE COMPLETION DELIVERY FOR LAUNCH
LAUNCH VERIFICAITON COMPLETION
DISPOSAL RETRIEVAL RE ENTRY READINESS

Figure 2.3: Example phasing of verification activities in the systems engineering lifecycle. Only the verification activities are shown here. A similar planning can be made for the validation activities as well. Source: [ECSS-E-HB-10-02A].

ance matrices, source code, mission diagrams, etcetera. A model is an artefact that represents (a part of) the system implementation, but is not the system itself. It can be either virtual, physical or both. Virtual models are intangible, and can for example be acoustic models, thermal models, software designs, hardware designs. Physical models are tangible, and can for example be hardware circuits, structural model, material models and scaled prototypes. A model can be both, if for example a software design is simulated on a hardware circuit (e.g. system-software co-design testing). Implementations are (parts of) the system itself, but not necessarily the system that will be flown. Multiple implementations may be produced, whereas one could be dedicated for extensive verification and validation, and another (lighter verified and validated) one could be used for the mission itself. This is to avoid possible wear on the latter system, which could be critical to the mission. Multiple implementations could also exist for keeping spares.

In the European space industry, four methods are distinguished for verifying and validating system artefacts. These are: review of design, inspection, analysis and testing. These following definitions are from the ECSS standard on verification [ECSS-E-ST-10-02C].

- Review of Design: A method typically used for verification where one checks whether "approved records or evidence that unambiguously show that the requirement is met. Examples of such approved records are design documents and reports, technical descriptions and engineering drawings". Requirements suitable for this method of verification are typically phrased existentially, e.g. "… shall have a monitoring device …".

- Inspection: This method consist of "visual determination of physical characteristics". Examples of physical characteristics are dimensions, but in practice may also include virtual characteristics, like software coding standards. Requirements suitable for this method are typically phrased physically or virtually quantifiable, e.g. "… shall be following C99 standard conventions …" or "… shall be 8 cm …".

- Testing: This method is the predominant method for verification and validation, and it is defined as "measuring product performance and functions under representative simulated environments". Requirements suitable for this requirement are typically quantifiable, e.g. performance or physical quantities, e.g. "… shall be less than 36 V at DC …" or "… shall sustain 40 Hz vibration for at least 3 seconds …". Typical tests for space systems are fit checks (for dimensions), pressure/leakage tests, electro-magnetic compatibility tests, mass checks (to stay within the range of launcher capabilities), shock tests, vibration tests, acoustic tests, separation tests, thermal cycling tests (for heat/cold differences) and vacuum tests. Venues containing the equipment to simulate space environments are for example available at ESTEC where these, typically costly, tests are performed. Also at a higher-level tests are conducted, often for system validation. Typically the philosophy at this stage is "test as you fly, fly as you test". Particular tests are end-to-end information system testing, where the compatibility of the spacecraft information systems with those of the ground segment is tested. Mission scenario tests are conducted, to test whether the flight hardware and software can execute missions under nominal conditions. Operations readiness tests are typically conducted for checking whether the ground segments work according to the mission plan, and a launch sequence is simulated to test whether the spacecraft is compatible with it. Additional stress testing and simulations are performed to check full system behaviour during simulated non-nominal conditions, and to inform the spacecraft operator on them and on possible mitigation approaches.

- Analysis: This method consists of "performing theoretical or empirical evaluation using techniques". Examples are statistical, quantitative and qualitative analyses. Requirements suitable for this method are typically phrased probabilistically and/or behaviourally, e.g. "…shall have a probability of …" or "…shall be designed to …". Certified analysis methods are typically preferred. With analysis, typically models are involved, and these need to be verified beforehand. The use of a particular analysis needs to be justified as well, e.g. whether it delivers accurate and adequate results and whether testing would not be cost-effective. Due to the theoretical or empirical nature of the analysis, boundary conditions and assumptions need to be clearly stated and the analytic uncertainty must be taken into account.

  There exists a plethora of analysis techniques, each historically developed for a particular engineering discipline and scope. In this dissertation, the ones that analyse technical risk are a core topic. Three dominant techniques for safety-critical systems are *fault tree analysis* [ECSS-Q-ST-40-12C], *failure modes, effects and criticality analysis* [ECSS-Q-ST-30-02C], *availability analysis* [ECSS-Q-ST-30-09C] and *probabilistic risk analysis*. These analyses are further elaborated in Chapter 4, where a formal approach towards them is defined.

2.1.5 Applicable Standards

For the European space industry, the management aspects of the space system life-cycle are defined in the M-10 series of the ECSS standards, whereas the engineering aspects are captured by the E-10 series. They ought not be followed literally, but tailored to the mission. The verification aspects are codified in [ECSS-E-ST-10-02C], and a set of guidelines that accompany these are defined in [ECSS-E-HB-10-02A]. As testing is the predominant method for verification and validation, a standard is dedicated to it [ECSS-E-10-03A]. In addition to these general verification and validation standards, there are several E-10 standards describing test methods tailored to specific subsystems.

Outside the European space industry, several standards exist that were formed through different heritages. NASA for example has its system engineering practices defined in its Systems Engineering Handbook [NASA/SP-2007-6105]. It is similarly constructed as its ECSS counterpart, and tends to differ mostly in terminology and the division of phases. With regard to verification and validation, NASA's requirements on that area are codified as part of the overall requirements on the systems engineering process [NPR 7123.1A]. With regard to safety specifically, NASA's counterpart of [ECSS-Q-ST-40C] is [NASA/SP-2010-580]. For probabilistic risk assessment NASA developed a procedures guide [NASA/SP-2011-3421] for its managers and practitioners. The US Department of Defense (DoD) maintains a set of practices [DoDI 5000.02] of which one part specifically reflects application to space [DTM-09-025].

2.2 Software Engineering

In the early days of the space-age spacecraft were highly mechanical and electronic in nature. Nowadays however, equipments tend to evolve to digital interfaces and the whole spacecraft becomes heavily computerised. Software plays an important role now. The most advanced spacecraft have software on-board that was compiled from millions lines of code (e.g. ESA's Automated Transfer Vehicle). The key reason is that digital interfaces, micro-processors and software can realise unprecedented functionality, allowing for highly demanding missions where automation and precision are driving mission success factors.

2.2.1 Software Uses for Space Missions

The acquisition of knowledge through data is a key objective for many space missions. The first uses of software were therefore in the data handling systems, both on-board and on the ground. On-board the spacecraft, the data handling system is responsible for managing, storing and processing all payload data, telecommands and telemetry. It is the interface with which the operator controls the spacecraft. The data handling system is at the same time also the interface through which mission data is retrieved. This data, e.g. from the Hubble telescope to Earth observation satellites, are vital and once transmitted to ground, are processed, stored, backed up, formatted and organised through a ground information system. Data integrity is of utmost importance here.

Spacecraft are furthermore equipped with a plenitude of control software. Launchers for example rely on guidance, navigation and control software for much of their mission lifetime. Satellites on the other hand require attitude and orbit control software. Software is also used for controlling power systems, thermal regulation systems and communication systems. Functional correctness of the software is of utmost importance for these applications.

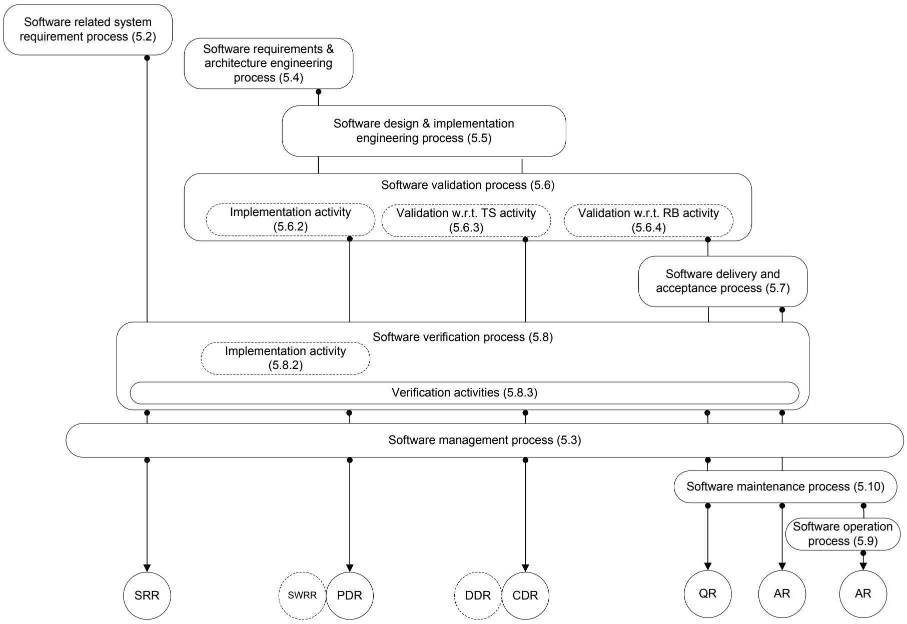

Software related system requirement process (5.2)
Software requirements & architecture engineering process (5.4)
Software design & implementation engineering process (5.5)
Software validation process (5.6)
Implementation activity (5.6.2)
Validation w.r.t. TS activity (5.6.3)
Validation w.r.t. RB activity (5.6.4)
Software delivery and acceptance process (5.7)
Software verification process (5.8)
Implementation activity (5.8.2)
Verification activities (5.8.3)
Software management process (5.3)
Software maintenance process (5.10)
Software operation process (5.9)
SRR
SWRR
PDR
DDR
CDR
QR
AR
AR

Figure 2.4: Typical software development process within the systems engineering life-cycle. Source: [ECSS-E-ST-40C].

The increase of a spacecraft's autonomy is typically realised through software. For deep-space missions, autonomy could be a necessity due to short communication windows and communication delays. Autonomy is also used to reduce costs, most particularly on the cost of operating the spacecraft. This holds in particular when constellations of spacecraft are deployed for the realisation of mission objectives. A special and prevalent type of autonomous functionality is that of fault management, also called *fault detection, isolation and recovery* (FDIR). This kind of functionality is predominantly realised using software, and manages the system during non-nominal operation and aims to preserve the system assets (e.g. crew, equipment) during anomalous conditions. These conditions are typically incurred by system failures or by the environment. Depending on the mission profile, this kind of software provides service to these conditions, with varying aims, like resuming nominal operations to mere system safing. The critical nature of fault management systems tend to make them worthwhile to model and analyse formally, and a section dedicated to such systems is found in Section 2.3.

2.2.2 Software Engineering Life-Cycle

The space software development process is depicted in Figure 2.4. It shows the software-related activities with respect to milestones of the systems engineering

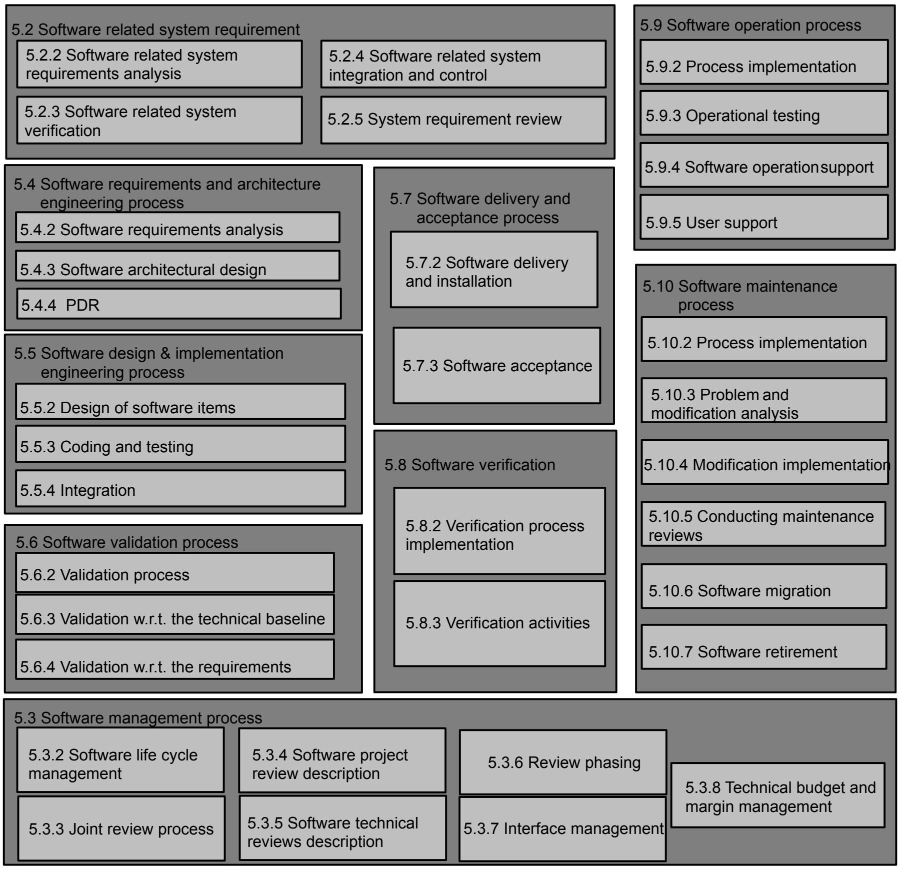

5.2 Software related system requirement
5.2.2 Software related system requirements analysis
5.2.4 Software related system integration and control
5.2.3 Software related system verification
5.2.5 System requirement review
5.9 Software operation process
5.9.2 Process implementation
5.9.3 Operational testing
5.9.4 Software operation support
5.9.5 User support
5.4 Software requirements and architecture engineering process
5.4.2 Software requirements analysis
5.4.3 Software architectural design
5.4.4 PDR
5.7 Software delivery and acceptance process
5.7.2 Software delivery and installation
5.7.3 Software acceptance
5.10 Software maintenance process
5.10.2 Process implementation
5.10.3 Problem and modification analysis
5.10.4 Modification implementation
5.10.5 Conducting maintenance reviews
5.10.6 Software migration
5.10.7 Software retirement
5.5 Software design & implementation engineering process
5.5.2 Design of software items
5.5.3 Coding and testing
5.5.4 Integration
5.8 Software verification
5.8.2 Verification process implementation
5.8.3 Verification activities
5.6 Software validation process
5.6.2 Validation process
5.6.3 Validation w.r.t. the technical baseline
5.6.4 Validation w.r.t. the requirements
5.3 Software management process
5.3.2 Software life cycle management
5.3.3 Joint review process
5.3.4 Software project review description
5.3.5 Software technical reviews description
5.3.6 Review phasing
5.3.7 Interface management
5.3.8 Technical budget and margin management

Figure 2.5: Typical software engineering activities in the space domain. Source: [ECSS-E-ST-40C].

life-cycle, e.g. systems requirements review, preliminary design review, critical design review, qualification review and acceptance review. The activities are numbered and refer to sub-activities depicted in Figure 2.5. It shows that the space software engineering life-cycle resembles common software engineering life-cycles with two exceptions. The main exception is the elaborate verification and validation process due to the inherent safety-critical nature of the system under development. The second exception is that the requirements often express interaction constraints with the system, i.e., the system-software co-engineering aspect. For regarding spacecraft, software on its own is rarely the end-product, but part of an overall system. The requirements defining the interaction between the system and software are thus crucial as input to software engineers, as well as the verification and validation of the combination. For this reason, software requirements engineering (and hence its development) is typically initiated before the system requirements review. If it were initiated even earlier, insufficient information about the system under development is available, rendering software design/requirements decisions to volatility. During qualification review, it is expected to have the software functioning. This means that the full space software development life-cycle is compressed in between. On top of that, the system requirements and design are still maturing in between, possibly affecting software requirements and design. Managing these co-engineering issues effectively is crucial for on-budget and on-schedule system delivery.

2.2.3 Software Considerations

The increased use of software results in an ever growing code base and code complexity. The first space systems in the 1960-1970 comprised only a few dozens lines of code. A decade later, the typical software size was around a few thousands lines. Nowadays, the software code base size goes into millions lines of code, and the trend is that this number grows exponentially [Dvo09]. The number of failures in aerospace systems related to software increased exponentially as well [Hec07].

The increase of the code size and complexity has spurred rapid innovations in this field. The consequence of this however is a less long-lived heritage of understanding costs, management and technology related to software. Furthermore, once a software technology has matured, its technological superiority is likely supplanted by a novel software technology. Novel (software) technology typically comes with a novel approach of its use. This makes management of software by similarity with previous software development projects less obvious.

Verifying software exhaustively is difficult and costly, due to the combinatorial nature of the offered software functionalities. Software is also easily modifiable, tempting engineers to restrict the design space on hardware while leaving design solutions open on the software-side. The margins left for software are however not physical in nature: whereas a physical component might allow for a few millimetres in size as a margin, a relaxation of a (combination of) software conditions easily leads to unaccounted software behaviours. Due to the software's flexibility, it has become increasingly acceptable for the final version of the software to be uploaded to the spacecraft while it is already launched into space. This provides for more time to have the software mature. Yet, it also adds more risk, as the final system configuration including its software is not verified and validated completely before launch, putting more pressure on getting the software correct.

Last but not least, software has become pervasive in every aspect of the space mission. Software is used in nearly all spacecraft's subsystems, necessitating complete and unambiguous system-software interaction requirements. In practice however, the system requirements are maturing during system development. Changes to those requirements can impact existing software interface requirements. To compensate for this, software engineers assume a wide-range system behaviours, whereas only a subset of it might be feasible. This especially holds for handling non-nominal operations of the spacecraft. On the other side, hardware engineers tend to assume software is correct by design, which leaves no margins for software glitches. On top of this, the software parts in the spacecraft's subsystems do not behave in isolation, but communicate heavily, allowing subsystems to account for other subsystems behaviour. This pervasive and interconnected nature require a holistic perspective of the system and the software, rather than reason about them in isolation. These issues are considered the prime system-software co-engineering issues.

2.2.4 Software Verification & Validation

Software is perceived as a component of the system to meet system objectives. From this perspective, the stakeholders of the software are not the same as that of the system. Instead, the stakeholders of the system are typically those who build the system. In this respect, software validation means checking whether it meets its requirements with respect to the system, i.e. checking the requirements baseline, and the requirements scoped purely to the software itself, i.e., the technical specification. Software validation is hence on checking the software behaviour.

Software verification therefore is not about checking its correctness with respect to "the" requirements, but rather to confirm whether specifications and inputs are adequate for every software development activity, and that the outputs are correct and consistent with the specifications and inputs [ECSS-E-ST-40C]. Software verification is hence on verifying artefacts of the software development process, rather than the software itself.

Planning

Before the system requirements review (SRR), the software related system requirement process (activity 5.2 of Figure 2.5) is started. The outputs of this process are the functional and performance requirements baseline (RB) for the software, and is the link between the system and the software. It captures the intents of the system engineer. During establishment of this baseline, early verification and validation activities are performed on both the system requirements and the requirements baseline.

After the system requirements review, software development starts with the definition of the software requirements, i.e. the technical specification (TS), and the software architecture. At this point, software validation is performed on the technical specification by checking it against the requirements baseline. Afterwards, during detailed design and implementation of the software, the validation against the requirements baseline is recurring, depending on the needed level of assurance. Once the technical specification has matured, which is typically in the early design and implementation process, the software is validated against the technical specification as well.

Due to the objective of software verification, the activity itself occurs throughout the majority of the software engineering life-cycle. It checks before and after the software activities whether their inputs and outputs are correct and consistent.

Methods

A plethora of methods exists for verifying and validating software, and these methods are typically used in a particular phase of the software engineering life-cycle. Depending on the situation, methods are also used outside their typical phase.

For early verification and validation of requirements, a typical method is test case generation. This forces system and software engineers to reason over the requirements and recognise weaknesses early. Especially the boundary conditions are of interest. Bidirectional traceability is used to ensure that all requirements and all components are interrelated. Rapid prototyping is used to deliver tangible results early, allowing for early feedback by stakeholders. In extremely critical cases, software requirements are simulated with hardware in the loop.

During architectural and detailed design, reuse analysis is used to determine suitability of adapting existing software to a new mission. Complexity measures can be employed, to check whether the software architecture and detailed design do not grow beyond the intended size. Viability analysis can be used to check whether the software architecture and/or detailed design provide sufficient functionality and information permitting the software validation engineer to enact on its activities. Prototyping for qualities does not focus on meeting observable behaviour, but on its inner qualities, like reliability and performance. Artefacts generated by this kind of prototyping are then subjected to analysis. Formal verification is used to check in a semantically rigorous way whether requirements by the technical specification and requirements baseline are met by the software architecture and design. It is the emphasized method in this discourse.

During the implementation process, the software becomes shaped in the form of source code. Inspections can then be performed, checking for example whether development conventions are met. Continuous integration is typically performed to check compatibility of separate software components, and understand early integration issues. Static analysers are used to check the source code on common semantic errors. Software operational resource assessment monitors the use of resources by the software, and checks whether they stay within specified bounds.

After implementation follows validation and qualification. Coverage analysis checks whether test cases exist for every path through the code base. In practice, full path-coverage is achieved only on parts of the software. Independent testers are invoked in this phase, as they tend to test the software with different assumptions that the software developer did. Differential testing can be performed if a reference software implementation exists, and results can be compared. Boundary testing is a simulative technique which drives the software implementation to boundary conditions, and observing how the software behaves. Non-nominal behaviour of the system is checked as well, and random reboots can be injected, as well as hardware failures to see how the software reacts to such situations. Multi-tasking overload tests can be conducted if the software is required to performs multiple activities in a scheduled order, and performance of these tasks are critical. Long-levity tests can be performed to see how software behaves over extended durations. Performance degradation due to memory leaks can for example be spotted this way.

Before flight, the software image is checked for corruptions and its authenticity. If the software is upgradable, the upgrade mechanism is tested too. The software is also validated through end-to-end information system testing, mission scenario testing and operational readiness tests.

The list of aforementioned methods is not exhaustive, and its use is not enforced. The choice and use of methods are dependent on the amount of acceptable risk.

2.2.5 Applicable Standards

In the European space industry, the standard on software in the ECSS engineering series [ECSS-E-ST-40C] describes the requirements and principles related to all software. Verification and validation aspects of software are part of the ECSS's software product assurance standard [ECSS-Q-ST-80C]. In addition to these standards, there are three ECSS handbooks related to software, namely one on the reuse of software [ECSS-Q-HB-80-01A], one on software process assessment and improvement [ECSS-Q-HB-80-02] and one on software dependability and safety [ECSS-Q-HB-80-03A].

At NASA, its equivalent of ECSS's software engineering standard is its software engineering requirements [NPR 7150.2A]. For software safety, NASA issues a guidebook [NASA-GB-8719.13] that complements the aforementioned requirements. The guidebook is not only on safety, but treats a wide area of topics that affect safety, like reuse and process improvement.

2.3 Fault Management Engineering

A huge engineering problem which inspires academia is that of fault management engineering. We define this branch of engineering as "the use of a cooperative design for flight and ground elements (including hardware, software, software, procedures, etc.) to detect and respond to perceived spacecraft faults ... It provides the ability for the spacecraft to detect, isolate and mitigate events that impact, or have the potential to impact nominal mission operations. This capability might be distributed across flight and ground systems, impacting hardware, software, and mission operation designs" [Fes+09]. It is traditionally perceived as an instance of software engineering, as the resulting engineering solutions are predominantly software-oriented. The methods offered by space software engineering however do not suffice. The current space software engineering approach is generally well-suited for engineering software for nominal behaviour. For non-nominal behaviour, which is covered by fault management systems, an additional set of unique problems upon those by software engineering are associated that have various dimensions. There is no convergence of terminology, little reusable heritage, no generalised life-cycle and processes, no distinct engineering responsibilities, little recognition and emphasis, no relevant metrics, no tools and no sound principles for fault management engineering. As a result, there are currently no mature industrial engineering approaches. In this section we discuss the most stringent problems, for which we argue that our methods and tools tackle a part of it. These, and the non-technical issues, which are left out of scope of this dissertation, are now an active field of research and development.

2.3.1 Terminology

As no common interpretation for fault management terminology has converged, we start with defining the terminology we use. It is aligned with the interpretation described in the recently developed taxonomy by NASA [Fes+09]. We believe those interpretations are the foundation for future fault management engineering efforts in the United States of America, and potentially globally.

A *failure* is an undesired effect, typically characterised by the violation of a nominal (system) requirement. The cause for a failure is called a *fault*. Failures and faults are typically intrinsically related through varying perspectives. For example a system can fail due to a faulty power supply. The failure of the power supply is due to a fault in both the batteries and the solar panels. A single event, e.g. the navigation computer is underpowered, is both a failure and a fault depending on the perspective, e.g. respectively the power subsystem and the system. Hence a failure is identified by the effect, whereas a fault is identified by the cause. Then there are *errors*. Those are discrepancies between a desired and estimated state. An error can be the fault for a failure, or a failure itself if the discrepancy is a violation of a nominal requirement. It can also be neither a fault nor a failure, since an error does not necessarily have undesired effects, nor does it is necessarily cause them. *Propagation* is another particular aspect to fault management. *Error propagation* is a trace of states in which each state contains an error. The trace captures a causal relation between those errors. *Failure propagation* on the other hand is defined from a requirements-perspective: it is a chain of faults and failures, where each fault is the cause for another failure. This is typically chained up to a system failure.

Fault management itself is often referred to by different terms, as Redundancy Management, Fault Protection Management, Health Management, Failure/Fault Detection, Isolation and Recovery (FDIR), Fault Detection, Response, Isolation and Recovery (FDRIR), Failure Detection, Isolation and Safing (FDIS). In this dissertation the term fault management will be used, even though failure management would be a more precise definition. This is intentionally, to have the terminology in this dissertation aligned with NASA's taxonomy.

2.3.2 Necessity of Fault Management

Fault management is an issue that emerged with particular mission profiles. For decades, most spacecraft did not go further than Earth orbit. Advanced forms of fault management were not considered for these kind of missions. The emphasis of these spacecraft was put on safing, which is the principle of ensuring the spacecraft's survival and allowing the operator to intervene. Such mechanisms have a relatively low complexity, as only failure detection mechanisms have to be put in place.

With deep-space missions, stringent issues with fault management became more apparent and trending. Operator intervention is not always an option due to the communication latency, as a spacecraft could fail permanently during the delay. Also with missions involving highly accurate orbits or trajectories, like navigation signals or particular scientific missions, may have higher demands on mitigation of failures. The fault management system in such systems typically acts as a passive controller of a hybrid system (the spacecraft's nominal subsystems). It is passive since is only intervenes when non-nominal behaviour is detected.

Inadequate fault management has led to mission failures in the past. Adequate fault management has also saved missions. It therefore is a necessity for a multitude of mission profiles.

2.3.3 Issues with Industry Practices

Issues with fault management engineering have been observed and recognised regardless whether the organisations involved were commercial or governmental. These issues, manifesting differently across organisations, appear to be systemic, and are due to an intertwining of issues. This section addresses the main technological, life-cyclic and organisational aspects of these issues. The contents is an extract from NASA's report on the first Fault Management Workshop [Fes+09] and infused with our experience from our research visits to ESA.

Technologically, fault management systems are complex, and they come with the issues of system-software co-engineering and more. The functionality provided by fault management systems is not isolated to a particular subsystem, but addresses all parts of the system and at all its levels, e.g. system, subsystem, equipment, component and parts. Yet, in the current practice, projects tend to focus on getting nominal behaviour right first, after which non-nominal behaviour is considered. The solutions are engineered as such that they can be "bolted" on the nominal system while meeting (or even exceeding) fault management requirements. More elegant solutions that require an intimate symbiosis of nominal and non-nominal design aspects are rejected, to avoid change of a reviewed and approved nominal system. Due to this, engineers typically have the gut-feeling that the resulting system is overly designed with fault tolerances that exceed the threshold of the mission risk profile. Furthermore, during operations, the added complexity due to over-design complicates satellite operation, especially when non-nominal situations arise. Capturing this degree of complexity is additionally made difficult due to the lack of standard terminology, interpretations and expressions means. Terminology and interpretations on fault management tend to be different throughout organisations, and even within organisations, conflicting interpretations exist for the same term. This manifests itself during milestone reviews and discussions. Additionally, there is no commonly agreed approach towards expressing fault management aspects, such as its architecture, behaviour, processes and analyses. The current expressions lack formalisation and rigour, adding more confusion that impose delays on the overall review process. Current methods also lack suitable tools that effectively support the design and analysis of fault management systems, resulting in a cumbersome process.

It is yet unclear how the fault management engineering life-cycle should look like, and how it should fit into the systems engineering life-cycle. There are no good generalised principles and no solid methodologies. This leads to ad-hoc engineering of solutions. There is also a lack of usable metrics to evaluate (intermediate) fault management artefacts on desirable qualities, like risk, complexity and performance. Especially metrics for functional aspects, like diagnosability, testability, usability and maintainability are lacking. Furthermore, the responsibilities and division of tasks are unclear, leading to a lack of ownership. Currently, the system engineer typically takes care of fault management issues as a side job. However due to the lack of priority for addressing non-nominal behaviour, fault management issues are not addressed at the right moment in the engineering life-cycle. The issues are then addressed too late. This impacts resource allocation, as unforeseen additional facilities, equipment and trained personnel have to be procured. This typically occurs during system integration and testing, where the overall system, including its fault management system, is pushed towards failure. More issues than initially estimated, budgeted and resolution time scheduled are the result. Generally, the fault management engineering life-cycle, as far as it is present, lacks continuous process improvement with respect to its verification and validation. Lessons learned earlier in the life-cycle are insufficiently leveraged to update verification and validation procedures in upcoming phases.

There are additionally issues with the perception of fault management within organisations. It typically lacks sufficient recognition at phases where it should. Especially at mission-level, fault management requirements and their impact to the system are not assessed and elevated to the appropriate levels of management. Furthermore, the issues are not tackled with the discipline and rigour it requires. This is due to the lack of trained personnel capable of dealing with fault management issues. Besides trained engineers, awareness is not fostered by managers, who tend to prioritise costs and risk different throughout the engineering life-cycle. Early in a project, the costs are prioritised, whereas later on, (avoidance of) risks are emphasised. This is especially visible when different organisations are involved in the project. Governmental organisations tend to design against possibility, whereas commercial organisations tend to design against probability. Varying attitudes towards fault management like this cause friction.

These issues combined can jeopardise the mission's readiness for launch. For planetary missions, launch opportunities are very hard deadlines. Budget overruns are caused to procure additional resources for overcoming the issues within time. Budget overruns however impact the ability to control costs and to fund new missions. It is therefore desirable to mature the engineering of fault management systems from all its dimensions. In this dissertation, we argue that our methods tackle a substantial part of the technological issues.

2.3.4 Applicable Standards

Fault management engineering has not matured yet to such a degree that standards on this topic exist. In existing ECSS standards, its issues are often briefly referred to as a concern to be dealt with. For example, in [ECSS-E-ST-10C], the existence of a fault management system (referred to as FDIR), is mandated. The same holds for [ECSS-E-HB-60A; ECSS-E-ST-20C; ECSS-Q-HB-80-04A]. In the standards on dependability [ECSS-Q-ST-30C] and modelling & simulation [ECSS-E-TM-10-21A], a few notes on fault management are mentioned that affect the topics covered by those standards. The space segment operability standard [ECSS-E-ST-70C] is more elaborate. It describes in six pages the FDIR-paradigm as an approach to fault management and a few technical issues that require consideration. The handbook on software dependability and safety [ECSS-Q-HB-80-03A] covers fault management issues as well, along with solutions, from a software perspective.

At NASA, fault management issues were encountered in nearly all their missions. In April 2008, a workshop was hosted where fault management practitioners came together, and formed a white paper on their findings [Fes+09]. Since then, resources were allocated to form a working group on fault management under the auspices of the Office of NASA's Chief Engineer, which led to the development of the NASA Fault Management Handbook [NASA-HDBK-1002]. At the time of writing this dissertation, we obtained an early draft of this handbook. It aims to systematically resolve many issues faced with fault management engineering, with in particular the life-cycle and organisational aspects.

3

Formal Architecture Modelling

To support system-software co-engineering, and especially fault management engineering, we argue that it is crucial to aim for completeness and unambiguity of the design artefacts (i.e. requirements and early architecture design) that constrain the system-software boundary. Furthermore, we argue that if they are specified with sufficient formality, then safety and dependability artefacts can be automatically derived from it. This enables early verification & validation results to system, software, RAMS and fault management engineers to quickly grasp the consequences of design choices at early engineering phases. Cornerstone of our approach is the modelling language, christened the System-Level Integrated Modelling Language (SLIM). It can be perceived as a formalised dialect of AADL, the Architecture and Analysis Design Language standardised by the Society of Automotive Engineers. It is suitable for modelling the system under development during early phase B, and especially before the system requirements review (SRR). Its formal character naturally pushes engineers to explore and reason critically over boundary conditions of the system's behaviour, motivating one to resolve issues early. This avoids the risk of underspecification and ambiguity that comes with it. Challenging is however to ensure that our modelling language is usable by focussing on architectural and behavioural detail without the need of capturing implementation-specific detail. Its notation needs to be attractive to engineers, and equally important, it needs to fit in the space systems engineering life-cycle.

3.1 Nominal Behaviour

The nominal part of SLIM is inspired by AADL 1.0 [SAE-AS5506], and in fact can be considered to be its extended subset. This section overviews the offered constructs. Later in the chapter, in Section 3.6, the differences between SLIM and AADL are discussed. Throughout this chapter, a running example of a battery-powered power system is used to exemplify the modelling language. Note that throughout this section, the exact syntax and syntactic restrictions are omitted, as this dissertation emphasises the semantics. A complete description of the syntax and its restrictions can be found in [Nol11b].

3.1.1 Structure

Components are first-class entities over which the engineers reason. In SLIM, they are reflected by a component type and one or multiple component implementations.

A component type describes the externally visible characteristics (i.e. interface) against which other components operate. In Listing 3.1, the `Battery` type is described as having three features, which are ports along which information is exchanged between components. The three features are an `empty` event, an input data port named `tryReset` and an output data port named `voltage` that has the initial value of 6.0. Ports are elaborated in Section 3.1.2.

Listing 3.1: Component type of a Battery.

```
system Battery
  features
    empty: out event port;
    tryReset: in data port bool default false;
    voltage: out data port real default 6.0;
end Battery;
```

Whereas component types describe *what* a component communicates, component implementations describe *how* a component communicates. It does this by defining its internal structure in terms of subcomponents, and how the interaction with the component's environment occurs along the ports defined by the component type. An example component implementation of a battery is shown in Listing 3.2. It contains details like modes which are discussed in subsequent subsections.

Part of the internal structure of a component are its subcomponents. These subcomponents can be either internal data (e.g. the subcomponent `energy` in Listing 3.2), or instantiations of other component implementations. The component-subcomponent relation gives rise to a component hierarchy, where the root component reflects the overall system, and the leaves the atomic components of which the system is composed. An example of this is shown in Listing 3.3, where a power system is composed of two battery components and a monitoring component.

In SLIM, component types, and their associated component implementations, are associated with a category which can be either hardware, software or a composite. Hardware categories are **processor**, **memory**, **bus**, and **device**. Software categories are **process**, **thread group**, and **thread**. A composite component is categorised as a **system**. This distinction can be used to clearly identify the system-software boundaries. Akin to AADL, SLIM allows to define bindings between the

Listing 3.2: Component implementation of a Battery.

```
system implementation Battery.Imp
  subcomponents
    energy: data continuous default 1.0;
  modes
    charged: activation mode while energy' = -0.02 and
                                   energy >= 0.2;
    depleted: mode while energy' = -0.03 and energy >= 0.0;
  transitions
    charged -[then voltage := 2.0* energy + 4.0]-> charged;
    charged -[reset when tryReset]-> charged;
    charged -[empty when energy = 0.2]-> depleted;
    depleted -[then voltage := 2.0 * energy + 4.0]-> depleted;
    depleted -[reset when tryReset]-> depleted;
end Battery.Imp;
```

system-software boundaries, indicating that a process is stored in a particular memory using the keywords `stored in`, that a process is running on a particular process using the keywords `running on`, or which bus is accessed by which hardware components using the keyword `accesses`.

3.1.2 Communication

Ports are the visible entities for other components to communicate with. In SLIM, we distinguish two types of ports, namely event ports and data ports. Event ports allow for rendez-vous communication between components. An example of this is the empty event in Listing 3.1. Event ports can be used as triggers between mode transitions, which are discussed in Section 3.1.3. Data ports allow for exposing information to other components. Ports are typed as either integer (i.e. `int`), real (i.e. `real`), enumeration (i.e. `enum`), or boolean (i.e. `bool`).

Ports are either incoming (i.e. `in event port` or `in data port`) or outgoing (i.e. `out event port` or `out data port`). Incoming ports represent the component's inputs, and the outgoing ports represent the component's outputs. This characterisation of ports affects the semantics of the enabledness of transitions (cf. Section 3.4) if they are labelled with an event.

Port connections can be used to relay events and data through components. A port connection between two incoming ports can for example represent the delegation of an event to another component, whereas a port connection between two outgoing ports typically represents exposure of information at lower levels of the system hierarchy to its higher levels. An example of this is shown in line 12 of Listing 3.3, where a `connections` part is specified. It is also possible to specify

Listing 3.3: Component type and implementation of a Power system.

```
system Power
  features
    alert: out data port bool observable;
end Power;

system implementation Power.Imp
    subcomponents
    batt1: system Battery.Imp accesses mybus in modes (primary);
    batt2: system Battery.Imp accesses mybus in modes (backup);
    mon: device Monitor.Imp accesses mybus;
    mybus: bus Bus;
  connections
    data port batt1.voltage -> mon.voltage in modes (primary);
    data port batt2.voltage -> mon.voltage in modes (backup);
    data port mon.alert -> alert;
    data port mon.alert -> batt1.tryReset in modes (primary);
    data port mon.alert -> batt2.tryReset in modes (backup);
  modes
    primary: activation mode;
    backup: mode;
  transitions
    primary -[batt1.empty]-> backup;
    backup -[batt2.empty]-> primary;
end Power.Imp;
```

a port connection between an outgoing port and an incoming port. This typically represents communication between components where the receiving side directly uses that information. It is possible to have multiple port connections targeting the same incoming event port, indicating a fan-in of events. This is however not possible for incoming data ports. It would be unclear which data source would be used at a particular moment. Fan-out is possible for both event and data ports. It is typically used to describe situations where information is broadcast to multiple components.

Data port connections relay information verbatim instantaneously. For many modelling cases, it is useful to relay *processed* information instantaneously. Flows are used in SLIM to achieve this. A flow is an assignment that is applied continuously. In Listing 3.4 for example, the flow defined on the outgoing port `alert` states that it becomes true whenever the voltage drops below 4.5. Only incoming data ports of the respective component, or the outgoing data ports of its subcomponents are allowed to be referred. The left-hand side must be an outgoing data port of the component, or an incoming data port of its subcomponents.

Listing 3.4: Component implementation of the monitor system.

```
device Monitor
    features
        voltage: in data port real;
        alert: out data port bool;
end Monitor;

device implementation Monitor.Imp
    flows
        alert := (voltage < 4.5);
end Monitor.Imp;
```

3.1.3 Modes

Modes serve two intertwined purposes. One, to provide an abstraction of behaviour of the concrete behaviour of a component and second, to provide anchors on which system topologies can be defined, such that these can be changed dynamically throughout the system's life-time.

Regarding behaviour, modes describe the possible states in which a component can reside. A change of component's mode is achieved by a transition, which is of the form `m1 -[e when g then f]-> m2`. The mode `m1` is the source mode, mode `m2` is the destination mode. If an event `e` is provided, the transition is either triggered externally if the event `e` is an incoming event port. If event `e` is an outgoing event port, then it must rendez-vous with an incoming event port. The guard `when g` is a boolean expression, which, when provided, only allows the transition to occur if g evaluates to true. If the transition occurs, one or more transition effects, i.e. `then` f, may occur. Effects are assignments, denoting an update of a data subcomponent or an outgoing data port. This is visible in the `transitions` part in Listing 3.2.

The example also shows the annotation of modes with hybrid behaviour. This behaviour is expressed through trajectory equations and mode invariants. The trajectory equation describes how the valuation of the variable changes over time. They are used for describing the evolution of a physical entity, like energy or temperature. For example the trajectory equation `energy' = -0.02` denotes that the `energy` variable decreases its value by `-0.02` every time unit. Mode invariants are bounds to this behaviour. For example, the mode invariant `energy >= 0.2` indicates that the system can only stay in the mode it is annotated to as long as `energy` is larger or equal than `0.2`. In SLIM, a data subcomponent of type `clock` is offered which is a special case of hybrid behaviour where the slope of the trajectory equation is simply `1`.

Behaviour and structure are interrelated through mode dependencies, which allow for dynamic reconfiguration of the system's component hierarchy. Using the clause `in modes`, port connections, flows and subcomponents are activated only in those particular modes. An example of this is shown in Listing 3.3, where the active battery's ports are rerouted based on the current mode. The `in modes` clause can also be used on flows. The absence of the `in modes` clause means that the element is active in all modes.

Using dynamic reconfiguration, components can be activated, deactivated and reactivated in their lifetime. If a component resets its state after reactivation, the keyword `activation` has to be used to denote the mode to which the component initialises and reinitialises. If a component does not reset after reactivation, the keyword `initial` has to be used to denote the mode at which the component initialises. In the latter case, upon reactivation, the component resumes from the mode in which it was active before deactivation.

3.1.4 Packages

SLIM offers packages to group components into a single namespace, allowing for the creation of libraries of reusable components. As they are a purely syntactic feature, and do not concern semantics, its discussion is omitted in this dissertation.

3.2 Erroneous Behaviour

Erroneous behaviour in SLIM is modelled using error model types and error model implementations, which is inspired by AADL's Error Model Annex 1.0 [SAE-AS5506/1]. The error behaviour of a complete system emerges from the combination of individual error component models. Failing components can affect other components due to their interaction, or because due to their shared hardware resources, which are defined through their bindings (cf. Section 3.1.1).

Error model types describe an interface in terms of error states and error propagations. Once an error model is associated with a nominal component, error states represent its current configuration with respect to the occurrence of errors. Error propagations are used to exchange error information between error models. This is useful when the nominal behaviour abstracts behaviour that could form a pathway for erroneous interaction. It is the modelling concept that explicitly introduces the possible erroneous interaction despite the abstracted nominal behaviour. An example of an error model is shown in Listing 3.5, line 3. The example shows an example of state history using the keyword **`initial`**. It is similar in concept to mode history (cf. Section 3.1.3).

Listing 3.5: Error model type for battery failures.

```
1  error model BatteryFailure
2    features
3      ok: initial state;
4      dead: error state;
5      resetting: error state;
6      batteryDied: out error propagation;
7  end BatteryFailure;
```

Error model implementations capture the behavioural aspect of the error model. An example of an error model implementation is shown in Listing 3.6. An error model implementation is defined by a (probabilistic) state machine whose states are declared in the error model type. Transitions between error states can be triggered by error events, **`reset`** events or error propagations. Error events are internal to the component and can reflect local faults or repair operations. They can be annotated with an occurrence rate. In SLIM, this is restricted to exponential distributions due to the underlying mapping to continuous time Markov chains. Take for example lines 10 and 11 in Listing 3.6. The error state `resetting` has two outgoing transitions, one that leads to the `ok` state due to a `work` event, which is annotated with the rate `0.2`. The other one goes to the state `dead` due to the event `fails`, which is annotated with the rate `0.8`. The probability that state `resetting` is left after waiting t time units is therefore $1 - e^{(0.2+0.8)t}$. The probability that it goes to state `ok` is $0.2/(0.2+0.8)(1 - e^{(0.2+0.8)t})$, whereas the probability that it goes to state `dead` is higher than for `ok`, namely $0.8/(0.2+0.8)(1 - e^{(0.2+0.8)t})$. Error models are not only about failures. Reset events (cf. line 9) are the built-in communication means between the nominal model and the error model. From the nominal model, these are typically sent when a successful repair occurred, and that the erroneous state needs to be updated accordingly.

Listing 3.6: Error model implementation for battery failures.

```
1  error model implementation BatteryFailure.Imp
2    events
3      fault: error event occurrence poisson 0.001;
```

```
    works: error event occurrence poisson 0.2;
    fails: error event occurrence poisson 0.8;
  transitions
    ok -[fault]-> dead;
    dead -[batteryDied]-> dead;
    dead -[reset]-> resetting;
    resetting -[works]-> ok;
    resetting -[fails]-> dead;
end BatteryFailure.Imp;
```

3.3 Model Extension

On its own, error models bear no tight relation with nominal models. The only and loose relation is the `reset` event. It can be used by the nominal model to indicate repairs. For the other direction, namely that erroneous behaviour affects nominal behaviour, a link is needed which is called a *fault injection*. It consists of three parts (s, d, a). The affected data element (i.e. data port or data subcomponent) in the nominal model is indicated by d, the fault expression is indicated by a, and the relevant error state is s. It means that whenever the error model is in error state s, the nominal value of d is overridden by a, the fault expression. Multiple fault injections are possible. With this approach, all possible data-failures can be captured.

Given the nominal models, the error models and the fault injections, an overall model is constructed called the *extended model*. The construction process itself is called model extension. The extended model is a single comprehensive model capturing both nominal and erroneous behaviour, and their interrelations. The principal idea is that the nominal and error models run concurrently. A state in the extended model consists of pairs of the nominal models and error states. Each transition in the extended model is due to a nominal transition or an error transition, or a synchronisation of both in case of a `reset` transition. To account for the fault injections, the nominal values of injected data elements are overridden by fault expressions if the error model is in the erroneous state specified by the fault injection.

More formally, the construction of the extended model occurs in two steps. First the error model component type and implementation are converted to a nominal component type and implementation. Then the nominal model is modified to incorporate the component types and implementations from the first step, and account for the failure effects.

3.3.1 Extended Error Model

Each error model type and implementation is converted to a nominal component type and implementation according to the following:

features Error events become outgoing event ports. Outgoing error propagations become outgoing event ports and incoming error propagations become incoming event ports.

modes Error states become modes. The **initial/activation** mark is preserved.

transitions Error transitions become mode transitions. For any mode without an outgoing **reset** transition, a self-loop is added with **#reset** as the trigger.

An example of these construction rules is shown in Listing 3.7. It is obtained by injecting the fault expression `voltage := 0` upon entering error state `dead`.

Listing 3.7: Part of the extended model describing the erroneous behaviour of battery.

```
system BatteryFailure
  features
    #reset: in event port;
    #fails: out event port;
    #batteryDied: out event port;
    #works: out event port;
    #fault: out event port
end BatteryFailure;

system implementation BatteryFailure.Implementation
  modes
    ok: initial mode;
    resetting: error mode;
    dead: error mode;
  transitions
    ok -[#fault]-> dead;
    dead -[#batteryDied]-> dead;
    dead -[#reset]-> resetting;
    resetting -[#works]-> ok;
    resetting -[#fails]-> dead;
    ok -[#reset]-> ok;
    resetting -[#reset]-> resetting;
end BatteryFailure.Implementation;
```

3.3.2 Extended Nominal Model

This nominal representation of the error model is added as a subcomponent to the nominal component, with additional modifications. The construction rules are:

`features` Nominal features are preserved. Ports are added due to error propagation. Incoming error propagations become incoming event ports and outgoing error propagations become outgoing event ports.

`subcomponents` Nominal subcomponents are preserved. A subcomponent named `._error` is added whose type is the nominal equivalent of the error model implementation.

`connections` Nominal port connections are preserved. Port connections are added by making error propagations between extended *subcomponents* explicit. This is due to either of the following:

- two subcomponents are bound to each other by **`accesses`**, **`running on`**, **`stored in`**, and one offers an outgoing error propagation and the other one accepts a matching incoming error propagation, or
- if two subcomponents are bound to each other by **`accesses`** on another common subcomponent, and one offers an outgoing error propagation and the other one accepts a matching incoming error propagation.

Also, error propagations in both directions between components in a super- and a subcomponent relation are made explicit.

`flows` Nominal flows are preserved, except for those with a fault injection. Those flows become the following:

- the fault expression, under the condition that the `_error` is in the injected error state.
- the nominal flow expression, under the condition that the `_error` is not in the injected error state.

`modes` Nominal modes are preserved.

`transitions` There are five types of transitions, namely

- Nominal transitions without fault injection. These are nominal transitions with their guards further constrained to non-injected error states.
- Nominal transitions with fault injection. These are nominal transitions with their guards further constrained to injected error states, and that the fault expression overrides the nominal expression of injected data elements.
- Error transitions without fault injection. These are nominal transitions that react to transitions leading to non-injected states from the error model.
- Error transitions with fault injection. These are nominal transitions that react to transitions leading to injected states from the error model. The fault expression overrides the nominal expression of injected data elements.

- Resets. These are nominal reset transitions that communicate explicitly with the resets of the error model.

The extended model shown in Listing 3.8 exemplifies the use of the aforementioned model extension rules.

Listing 3.8: Extended model of a battery.

```
system Extended_batt1_Battery
  features
    voltage: out data port real default 6.0;
    #batteryDied: out event port;
    empty: out event port;
end Extended_batt1_Battery;

system implementation Extended_batt1_Battery.Imp
  subcomponents
    energy: data continuous default 1.0;
    _error: system BatteryFailure.Implementation;
  connections
    event port _error.#batteryDied -> #batteryDied;
  modes
    depleted: mode while energy' = -0.03 and energy >= 0.0;
    charged: activation mode while energy' = -0.02 and
                                   energy >= 0.2;
  transitions
    -- nominal transitions
    charged -[when _error.state != dead
              then voltage := 2.0 * energy + 4.0]-> charged;
    charged -[empty when energy = 0.2 and
                            _error.state != dead]-> depleted;
    depleted -[when _error.state != dead
               then voltage := 2.0 * energy + 4.0]-> depleted;
    -- nominal transitions with fault injection
    charged -[when _error.state = dead then voltage := 0]-> charged;
    charged -[empty when energy = 0.2 and _error.state = dead
                    then voltage := 0]-> depleted;
    depleted -[when _error.state = dead
               then voltage := 0]-> depleted;
    -- error transitions
    depleted -[_error.#works
               when _error.state = resetting]-> depleted;
    charged -[_error.#works
              when _error.state = resetting]-> charged;
    -- error transitions with fault injections
    depleted -[_error.#fault when _error.state = ok
                              then voltage := 0]-> depleted;
    charged -[_error.#fault when _error.state = ok
                             then voltage := 0]-> charged;
    depleted -[_error.#batteryDied when _error.state = dead
                                     then voltage := 0]-> depleted;
    charged -[_error.#batteryDied when _error.state = dead
                                    then voltage := 0]-> charged;
    depleted -[_error.#fails when _error.state = resetting
                              then voltage := 0]-> depleted;
    charged -[_error.#fails when _error.state = resetting
                             then voltage := 0]-> charged;
    -- resets
    charged -[_error.#reset when tryReset]-> charged;
    depleted -[_error.#reset when tryReset]-> depleted;
end Extended_batt1_Battery.Imp;
```

3.4 Formal Semantics

The formal semantics maps a SLIM specification to an automata-like formalism, namely a network of event-data automata. Each event-data automaton is the representation of a SLIM component. From these automata-like formalisms, a transition system can be spawned, representing its behaviour in a formal way.

3.4.1 Event-Data Automata

This subsection introduces the definition of an event-data automaton (i.e. EDA), and defines how a single SLIM component is mapped to it. Afterwards its semantics are defined as a transition system. The battery component of the power system example is used to exemplify these notions.

Definition

An *event-data automaton* is a tuple of the form

$$\mathfrak{A} = (M, m_0, X, v_0, \chi, \varphi, E, \longrightarrow)$$

where

- M is a finite set of *modes*.
- $m_0 \in M$ denotes the *starting mode*.
- X is a finite set of *variables*, partitioned into

  - *input variables, IX,*
  - *output variables, OX,*
  - *local variables, LX.*

- $v_0 \in V_X$ is the *initial valuation* where V_X denotes the set of all *valuations*, that is, partial functions that assign values to the elements of X.

- $\chi : M \to (V_{LX} \to \mathbb{B})$ specifies the *mode invariants* (where we assume that $\chi(m_0)(v_0|_{LX}) = \top$).

- $\varphi : M \to (LX \to \mathbb{R})$ specifies the *trajectory equations* by associating with each local variable its derivative in the current mode.

- E is a finite set of *events*, partitioned into

  - *input events, IE,*
  - *output events, OE.*

- $\longrightarrow \subseteq M \times E_\tau \times (V_X \to \mathbb{B}) \times (V_X \to V_X) \times M$ is a finite *(mode) transition relation* where $E_\tau := E \cup \{\tau\}$. The τ event indicates the absence of an event trigger. A transition is represented in the form $m \xrightarrow{e,g,f} m'$, and e, g, and f which are respectively called the *trigger*, the *guard*, and the *effect*. Here f is only allowed to modify output and local variables, that is, $f(v)(x) = v(x)$ for each $v \in V_X$ and $x \in IX$.

For the sake of generality, we do not restrict the value ranges of the partial functions in V_X. In our concrete setting, valuations assign, e.g., Boolean, integer, and real values to the variables in X.

Furthermore, we assume that each invariant $\chi(m)$ with $m \in M$ is given by a Boolean expression over local variables where each arithmetic subexpression is linear. Also trajectory equations are only defined for local variables and their evolution is described by linear functions. If x is a discrete variable, then $\varphi(m)(x) = 0$; if x is a *clock*, then $\varphi(m)(x) = 1$; otherwise x is a *continuous* variable. Our semantics can support more involved trajectory equations, but we restrict ourselves to constant slopes for feasibility of conducting analyses.

Mapping from SLIM

A SLIM component has a straightforward representation as an EDA. It is based on the following associations:

- the modes M are mapped from the `modes` part in a SLIM component implementation:

  `modes` … *m*: *tp* `mode` …

  If that occurs, then

  - $m \in M$, and
  - $m_0 = m$ if $tp \in \{$**initial**, **activation**$\}$.
- the variables X comprise $IX \cup OX \cup LX$, which are obtained from the data ports and the data subcomponents:
  - any data port from **features** part of the SLIM component type:

    features ... p: *io* **data port** ... **default** a;

    If $io =$ **in**, then $p \in IX$. If $io =$ **out**, then $p \in OX$.
  - any data subcomponent from the `subcomponent` part of the SLIM component implementation:

    subcomponents ... sc: **data** ... **default** a;

    Then $sc \in LX$.
- the initial valuation v_0 of variables is defined through the **default** keyword. For data ports, as described with the previous point, $v_0(p) = a$. For data subcomponents, it means that $v_0(sc) = a$. Otherwise $v_0(p)$ is undefined in the EDA, although it has to be defined through other ports in the NEDA (cf. Section 3.4.2).
- the mode invariants χ and trajectory equations φ are defined through the **while** part of a mode declaration:

  modes ... m: ... **mode while** iv; ...

  The expression iv is composed of conjuncted subexpressions $iv_1 \wedge \ldots \wedge iv_n$. If a subexpression iv_i is a trajectory equation, i.e. iv_i is `sc' = s`, then $\varphi(m)(sc) = s$. The remaining subexpressions must be Boolean expressions that, when conjuncted, represent the mode invariant $\chi(m)$.
- the set of events E comprises of $IE \cup OE$, where
  - an incoming event port in the **features** part of the component type:

    features ... p: **in event port**; ...

    result to $p \in IE$. For each subcomponent sc of the component, its outgoing event ports are also included in IE.
  - an outgoing event port in the **features** part of the component type:

    features ... p: **out event port**; ...

    results to $p \in OE$. For each subcomponent sc of the component, its incoming event ports are also included in OE.

The component's subcomponent ports are used to enable event communication between the component and its (active) subcomponent(s).

- the transition relation $\longrightarrow$ emerges from the **`transitions`** part in the SLIM specification. More specifically, a transition of the form

  `transitions...m -[e when g then f]-> m'`

  results to a transition in $(m, e', g', f', m') \in \longrightarrow$. The event e is mapped to e' if it is present, otherwise $e' = \tau$. The guard g is mapped to g' if it is present, otherwise $g' = \top$. The transition effects $f'(v) = v'$ are defined as follows:

  - for each $d \in IX$ it holds that $v'(d) = v(d)$.
  - for each $d \in OX$ it holds that

    $$v'(d) := \begin{cases} [\![a]\!](v) & \text{if f contains assignment } d := a \\ v(d) & \text{otherwise} \end{cases}$$

  - for each $d \in LX$ it holds that

    $$v'(d) := \begin{cases} [\![a]\!](v) & \text{if f contains assignment } d := a \\ v_0(d) & \text{else if } d \text{ inactive in mode } m \\ v(d) & \text{otherwise} \end{cases}$$

  The notation $[\![a]\!](v)$ means a single value that is the result of an evaluation of expression a using the valuation mapping v.

Transition System

The operational semantics of an EDA is given as a labelled transition system whose states, called *configurations*, are pairs of modes and valuations. Transitions either model the passage of time, involving an update of the local variables (but no mode change), or are internally triggered by events, including the internal event τ. The second case requires the guard of the respective transition to be enabled, and then modifies the valuation of the variables according to the transition effect. It is assumed that the processing of events is instantaneous, i.e., takes no time.

The definition of the semantics employs the following notation. Given a valuation $v \in V_X$, a time delay $t \in \mathbb{R}_{>0}$, and a mapping $\varphi : LX \to \mathbb{R}$ of derivatives, the notation $v + t \cdot \varphi$ denotes the corresponding temporal modification of the local variables, that is, for each $x \in X$,

$$(v + t \cdot \varphi)(x) := \begin{cases} v(x) + t \cdot \varphi(x) & \text{if } x \in LX \\ v(x) & \text{otherwise} \end{cases}$$

Given these definition, we formally give the *semantics* of an EDA by a labelled transition system

$$(Cnf, \kappa_0, L, \longrightarrow)$$

with

- the set of *(local) configurations* $\mathit{Cnf} := M \times V_X$,
- the *initial configuration* $\kappa_0 := (m_0, v_0) \in \mathit{Cnf}$,
- the set of *transition labels* $L := \mathbb{R}_{>0} \cup E_\tau$, and
- the *(local) transition relation* $\longrightarrow \subseteq \mathit{Cnf} \times L \times \mathit{Cnf}$, given by either a
  - *time transition*:
    $$(m, v) \xrightarrow{t} (m, v + t \cdot \varphi(m))$$
    if $t \in \mathbb{R}_{>0}$ and the invariant of mode m, $\chi(m)(v + t' \cdot \varphi(m))$ holds for any $t' \in [0, t]$.
  - *internal or event transition*:
    $$(m, v) \xrightarrow{e} (m', f(v))$$
    if a transition triggered by $e \in E_\tau$ is enabled in mode m and also the invariant of target mode m' is valid after applying the transition effect, that is, there exists $m \xrightarrow{e,g,f} m'$ such that $g(v) = \top$ and $\chi(m')(f(v)|_{LX}) = \top$.

Example

The battery example in Listing 3.2 yields the following event-data automaton $\mathfrak{A} = (M, m_0, X, v_0, \chi, \varphi, E, \longrightarrow)$ where

$$\begin{array}{rcl}
M & = & \{\texttt{charged}, \texttt{depleted}\} \\
m_0 & = & \texttt{charged} \\
X & = & \underbrace{\{\texttt{tryReset}\}}_{IX} \cup \underbrace{\{\texttt{voltage}\}}_{OX} \cup \underbrace{\{\texttt{energy}\}}_{LX} \\
v_0 & = & [\texttt{tryReset} \mapsto \bot, \texttt{voltage} \mapsto 6.0, \texttt{energy} \mapsto 1.0] \\
\chi(\texttt{charged})(v) & = & (v(\texttt{energy}) \geq 0.2) \\
\chi(\texttt{depleted})(v) & = & (v(\texttt{energy}) \geq 0.0) \\
\varphi(\texttt{charged})(\texttt{energy}) & = & -0.02 \\
\varphi(\texttt{depleted})(\texttt{energy}) & = & -0.03 \\
E & = & \underbrace{\emptyset}_{IE} \cup \underbrace{\{\texttt{empty}\}}_{OE} \\
\longrightarrow(\texttt{charged}) & = & \{(\tau, \top, v[\texttt{voltage} \mapsto 2.0 \cdot \texttt{energy} + 4.0], \texttt{charged}), \\
 & & \ (\mathbf{reset}, v(\texttt{tryReset}), v, \texttt{charged}), \\
 & & \ (\texttt{empty}, (v(\texttt{energy}) = 0.2), v, \texttt{depleted})\} \\
\longrightarrow(\texttt{depleted}) & = & \{(\tau, \top, v[\texttt{voltage} \mapsto 2.0 \cdot \texttt{energy} + 4.0], \texttt{depleted}), \\
 & & \ (\mathbf{reset}, v(\texttt{tryReset}), v, \texttt{depleted})\}
\end{array}$$

The operational semantics of the EDA that is obtained from the `Battery` specification gives rise to the trace shown in Figure 3.1. Here we denote a configuration by a pair of the form $\langle m, v\rangle$ where $m \in M$ and $v \in V_X$.

$$\begin{array}{c}
\langle\ \texttt{charged}, [\texttt{energy} \mapsto\ 1.0, \texttt{tryReset} \mapsto \bot, \texttt{voltage} \mapsto 6.0]\rangle \\
\downarrow 30.0 \\
\langle\ \texttt{charged}, [\texttt{energy} \mapsto\ 0.4, \texttt{tryReset} \mapsto \bot, \texttt{voltage} \mapsto 6.0]\rangle \\
\downarrow \tau \\
\langle\ \texttt{charged}, [\texttt{energy} \mapsto\ 0.4, \texttt{tryReset} \mapsto \bot, \texttt{voltage} \mapsto 4.8]\rangle \\
\downarrow 10.0 \\
\langle\ \texttt{charged}, [\texttt{energy} \mapsto\ 0.2, \texttt{tryReset} \mapsto \bot, \texttt{voltage} \mapsto 4.8]\rangle \\
\downarrow \tau \\
\langle\ \texttt{charged}, [\texttt{energy} \mapsto\ 0.2, \texttt{tryReset} \mapsto \bot, \texttt{voltage} \mapsto 4.4]\rangle \\
\downarrow \texttt{empty} \\
\langle \texttt{depleted}, [\texttt{energy} \mapsto\ 0.2, \texttt{tryReset} \mapsto \bot, \texttt{voltage} \mapsto 4.4]\rangle \\
\downarrow 5.0 \\
\langle \texttt{depleted}, [\texttt{energy} \mapsto 0.05, \texttt{tryReset} \mapsto \bot, \texttt{voltage} \mapsto 4.4]\rangle \\
\downarrow \tau \\
\langle \texttt{depleted}, [\texttt{energy} \mapsto 0.05, \texttt{tryReset} \mapsto \bot, \texttt{voltage} \mapsto 4.1]\rangle \\
\downarrow \vdots
\end{array}$$

Figure 3.1: An example trace of the Battery component.

3.4.2 Network of Event-Data Automata

The global system behaviour of a SLIM specification emerges from the interaction between EDAs. This interaction is highly dynamic as local transitions can cause subcomponents to become (in-)active, and can change the topology of event and data port connections and flows. On the level of the formal semantics, which is given as a network of EDAs, this means that both the activation of the component EDAs and their interconnections depend on the modes of the individual EDAs. In the sequel, let $[n]$ for a natural n denotes the set $\{1,\ldots,n\}$ and $\dashrightarrow$ denotes a partial function.

Definition

Formally, a *network of event-data automata (NEDA)* is a tuple of the form

$$\mathfrak{N} = ((\mathfrak{A}_i)_{i\in[n]}, \alpha, EC, DD)$$

where

- each $\mathfrak{A}_i$ is an EDA of the form $\mathfrak{A}_i = (M_i, m_0^i, X_i, v_0^i, \chi_i, \varphi_i, E_i, \longrightarrow_i)$ with $i \in [n]$,

- $\alpha : M \to 2^{[n]}$ is the *activation mapping* where $M := \prod_{i=1}^{n} M_i$ denotes the set of *global modes*,
- $EC : M \to (\{i.oe \mid i \in [n], oe \in OE_i\} \times \{j.ie \mid j \in [n], ie \in IE_j\})$ is the *event connection mapping*, and
- $DD : M \to (\{i.x \mid i \in [n], x \in IX_i \cup OX_i\} \dashrightarrow \{j.a \mid j \in [n], a \in Exp(IX_j) \cup OX_j\})$ is the *data dependence mapping* where $Exp(IX_j)$ denotes the set of all expressions over IX_j.

The activation mapping α specifies the active (sub)components in a given global mode. In other words, $\alpha(m_1, \ldots, m_n)$ is the set of components that are active in mode $(m_1, \ldots, m_n)$, where m_i denotes the local mode of component i.

The mapping EC provides mode-dependent interdependencies between event ports. If $(i.oe, j.ie) \in EC(m_1, \ldots, m_n)$, then in mode $(m_1, \ldots, m_n)$, the outgoing event port oe of component i is connected to the incoming event port ie of component j. Here, EC is a binary relation (and not a function) as our specification language supports fan-in for event ports.

The mapping DD provides mode-dependent interdependencies between data ports. If $DD(m_1, \ldots, m_n)(i.x) = j.a$, then either data port a of component j is connected to data port x of component i in mode $(m_1, \ldots, m_n)$, or there is a flow relation for component $i = j$ in mode m_i which defines the value of data port x by the expression a. Both out-to-out, out-to-in, and in-to-in connections are possible. In the second case, x must be an outgoing port and a must be an expression over the incoming ports of the same component. Here, DD is a proper (partial) mapping as our specification language does not support fan-in for data ports since this would not yield unique values.

Mapping from SLIM

The mapping of a full SLIM specification to a network of event-data automata accounts for dynamic reconfigurations using the `in mode` clauses in the SLIM specification and for event communication across the component hierarchy. The latter can occur by chaining event port connections, so that events from a component can synchronise with transitions that are beyond their direct supercomponent and subcomponents.

Formally, given the collection of components in the SLIM specification, the association of a corresponding NEDA,

$$\mathfrak{N}_S = ((\mathfrak{A}_i)_{i \in [n]}, \alpha, EC, DD),$$

can be defined as follows:

- each $\mathfrak{A}_i$ with $i \in [n]$ is an EDA representing a component instance in the component hierarchy. It is constructed according to the description in Section 3.4.1,

- the activation mapping $\alpha : M \to 2^{[n]}$ is defined recursively from the root component. For each $(m_1,\ldots,m_n) \in M$,
  - the root component, designated as 1 is always active, more formally $1 \in \alpha(m_1,\ldots,m_n)$, and
  - a subcomponent *sc* designated active of an active component c_i is also active, i.e. for the following piece of SLIM specification associated with $\mathfrak{A}_i$:

    subcomponents … *sc*: …**in modes** (*AM*);…

    where *AM* is a subset of modes, i.e., $AM \subseteq M_i$ of EDA i. Let $j \in [n]$ be the EDA associated with *sc* and $i \in \alpha(m_1,\ldots,m_n)$ then $j \in \alpha(m_1,\ldots,m_n)$.
- the event port connections $EC(m_1,\ldots,m_n)$ are determined by event port connections occurring in the **connections** part of a component implementation, i.e.:

  connections … **event port** $c_1.p_1$`->` $c_2.p_2$ **in modes** (*AM*);

  If the referenced port occurs in the current component, then the subcomponent identifier, e.g. c_1 or c_2 is absent. From the port connections, which are only active in the set of modes given by *AM*, a part of *EC* is built up by:

  $$\{(i.op, j.ip) \mid i,j \in [n], op \in OE_i, ip \in IE_j, (c_i.op, c_j.ip) \in ECon^+\}$$

  Active event port connections can be chained, so that transitions can synchronise with those of other components beyond its direct super-component or subcomponents. Such a chain is captured by $ECon^+$ where $(c_i.op, c_j.ip) \in ECon^+$ means that, in the global mode $(m_1,\ldots,m_n)$, there is a chain of connections from the output port *op* of component c_i to the input port *ip* of component c_j, in the order

  1. (zero or more) out-to-out event port connections,
  2. (exactly one) out-to-in event port connection, and
  3. (zero or more) in-to-in event port connections.

  There are also implicit event-port connections between super- and subcomponents. Two cases are possible, namely that a super-component refers to an incoming event as a transition trigger or to an outgoing event. Both are captured by respectively

  $$\{(i.(sc.ip), j.ip) \mid i,j \in [n], j \in \alpha(m_1,\ldots,m_n), c_i.sc = c_j, sc.ip \in OE_i\}$$

  and

  $$\{(j.op, i.(sc.op)) \mid i,j \in [n], j \in \alpha(m_1,\ldots,m_n), c_i.sc = c_j, sc.op \in IE_i\}$$

  The three sets unioned together form *EC*.

- any value of an incoming or outgoing data port is defined as follows for each $(m_1, \dots, m_n) \in M$, $i \in [n]$, and $x \in IX_i \cup OX_i$,

$$DD(m_1, \dots, m_n)(i.x) := \begin{cases} j.y & \text{if } (y, sc.x) \in DCon(c_j, m_j) \text{ and } c_j.sc = c_i \\ & \text{or } (sc_1.y, sc_2.x) \in DCon(c_k, m_k) \text{ and} \\ & \quad c_k.sc_1 = c_j, c_k.sc_2 = c_i \\ & \text{or } (sc.y, x) \in DCon(c_i, m_i) \text{ and } c_i.sc = c_j \\ i.a & \text{if } (a, x) \in Flw(c_i, m_i) \\ \text{-} & \text{undefined otherwise} \end{cases}$$

In the above, the set $DCon(c_j, m_j)$ describes the data port connections occurring in component c_j and that are active in mode m_j. They are obtained from the SLIM component implementation via

connections ... **data port** $c_1.p_1$-> $c_2.p_2$ **in modes** (AM);

where $m_j \in AM$ and c_j is the component in which the data port connection is set between two subcomponents of component c_j. Data port connections can also occur from super-components to subcomponents, which would be then as follows:

connections ... **data port** p_1-> $c_2.p_2$ **in modes** (AM);

or from subcomponents to super-components:

connections ... **data port** $c_1.p_1$-> p_2 **in modes** (AM);

The set $Flw(c_i, m_j)$ describes the flows occurring in component c_j and are active in mode m_j. They are obtained from the SLIM component implementation via

flows ... $d := a$ **in modes**(AM);

Each flow $(a, d) \in Flw(c_i, m_j)$ is determined by the flow expression a and the outgoing data port d for which it is defined. It is active if $m_j \in AM$. It is undefined in all other cases, in which the data port is assigned by transitions effects or by a data port connection.

Transition System

The *semantics* of a NEDA is given by the labeled transition system

$$(Cnf, \kappa_0, L, \Longrightarrow)$$

which is defined in terms of the local transition systems $(Cnf_i, \kappa_0^i, L_i, \longrightarrow_i)$ with $i \in [n]$ of the constituent EDAs as follows

- the set of *(global) configurations* is given by $Cnf = \prod_{i=1}^{n} Cnf_i$,

- the *initial configuration* is $\kappa_0 = (\kappa_0^1, \ldots, \kappa_0^n)$,
- the set of *transition labels* is $L = \mathbb{R}_{>0} \cup \{\tau\}$, and
- the *(global) transition relation,* $\Longrightarrow \subseteq \mathit{Cnf} \times L \times \mathit{Cnf}$, is given by
  - *time transition*:

    $$\kappa = (\kappa_1, \ldots, \kappa_n) \overset{t}{\Longrightarrow} (\kappa'_1, \ldots, \kappa'_n)$$

    if

    · $t \in \mathbb{R}_{>0}$ and

    · all active EDAs are involved in the time step: for each $i \in [n]$, $\kappa_i \xrightarrow{t} \kappa'_i$ if $i \in \alpha(\mathit{mod}(\kappa))$, and $\kappa'_i := \kappa_i$ otherwise.

    The auxiliary function *mod* is defined on page 48.
  - *internal transition*:

    $$\kappa = (\kappa_1, \ldots, \kappa_n) \overset{\tau}{\Longrightarrow} \mathit{cns}_\kappa(\mathit{nxt}(\kappa, \{(i, \kappa'_i)\}))$$

    if

    · the ith EDA is active and can perform an internal step: there exists $i \in \alpha(\mathit{mod}(\kappa))$ and $\kappa'_i \in \mathit{Cnf}_i$ such that $\kappa_i \xrightarrow{\tau}_i \kappa'_i$.

    The auxiliary functions *cns* and *nxt* are defined on page 49.
  - *multiway communication transition*:

    $$\kappa = (\kappa_1, \ldots, \kappa_n) \overset{\tau}{\Longrightarrow} \mathit{cns}_\kappa(\mathit{nxt}(\kappa, \{(j, \kappa'_j) \mid j \in J \cup \{i\}\}))$$

    if

    · the i^{th} EDA is active and offers an output transition: there exists $i \in \alpha(\mathit{mod}(\kappa))$, $\mathit{oe} \in \mathit{OE}_i$, and $\kappa'_i \in \mathit{Cnf}_i$ such that $\kappa_i \xrightarrow{\mathit{oe}}_i \kappa'_i$, and

    · there is at least one active neighbor EDA that offers a corresponding input transition via an event port connection:

    $$\begin{array}{rcl} J & = & \{j \in \alpha(\mathit{mod}(\kappa)) \setminus \{i\} \mid \\ & & \quad \text{exists } \mathit{ie} \in \mathit{IE}_j \text{ s.t. } (i.\mathit{oe}, j.\mathit{ie}) \in \mathit{EC}(\mathit{mod}(\kappa)) \\ & & \quad \text{and } \kappa_j \xrightarrow{\mathit{ie}}_j \kappa'_j\} \\ & \neq & \emptyset. \end{array}$$

The definition employs the following auxiliary functions:

- $\mathit{mod} : \mathit{Cnf} \to M$, extracting the mode information from a given global configuration:

  $$\mathit{mod}(\kappa_1, \ldots, \kappa_n) := (\mathit{mod}(\kappa_1), \ldots, \mathit{mod}(\kappa_n)) \text{ with } \mathit{mod}(m, v) := m.$$

- $nxt : Cnf \times 2^{\bigcup_{i\in[n]}\{i\}\times Cnf_i} \rightarrow Cnf$, which reflects the impact of mode transitions occurring in the constituent EDAs, taking the current glocal configuration (first parameter) and the new local configurations (second parameter) into account. This impact is defined as follows:
  - Each EDA occurring in the set enters the new configuration.
  - Next, the impact of the mode transitions on the affected components is determined as follows. Each component that is re-activated in the transition (that is, it is inactive in the source mode but active in the target mode) and that does not support mode history (that is, its starting mode carries the `activation` attribute) is restarted. This means that it enters its starting mode, and that its data elements obtain their default values.

  Formally, this is described as follows.

  $$nxt(\kappa, N) := restart_\kappa(config(\kappa, N))$$

  where

  $$\begin{aligned} config(\kappa, \emptyset) &:= \kappa \\ config((\kappa_1, \ldots, \kappa_i, \ldots, \kappa_n), \{(i, \kappa_i')\} \cup N) &:= config((\kappa_1, \ldots, \kappa_i', \ldots, \kappa_n), N) \\ restart_\kappa(\kappa_1', \ldots, \kappa_n') &:= (\kappa_1'', \ldots, \kappa_n'') \end{aligned}$$

  with

  $$\kappa_i'' := \begin{cases} \kappa_0^i & \text{if } i \in \alpha(mod(\kappa_1', \ldots, \kappa_n')) \setminus \alpha(mod(\kappa)) \text{ and} \\ & mod(\kappa_0^i) \text{ is an } \texttt{activation} \text{ mode} \\ \kappa_i' & \text{otherwise} \end{cases}$$

- $cns_\kappa : Cnf \rightarrow Cnf$, making a global configuration consistent by taking the (unique) solution of the equation system that is implied by the data dependence mapping. In addition, input or output variables that have been disconnected in the transition (that is, the variable occurs as a target in the data dependence relation of the old mode but in no data dependence of the new mode) are reset to their default values:

  $$cns_\kappa((m_1, v_1), \ldots, (m_n, v_n)) := ((m_1, v_1'), \ldots, (m_n, v_n'))$$

  if, for each $i \in [n]$ and $x \in IX_i \cup OX_i$,

  $$v_i'(x) = \begin{cases} [\![a]\!](v_j') & \text{if } DD(m_1, \ldots, m_n)(i.x) = j.a \\ v_0^i(x) & \text{if } DD(m_1, \ldots, m_n)(i.x) \text{ is undefined and} \\ & \quad DD(mod(\kappa))(i.x) \text{ is defined} \\ v_i(x) & \text{if } DD(m_1, \ldots, m_n)(i.x) \text{ is undefined and} \\ & \quad DD(mod(\kappa))(i.x) \text{ is undefined} \end{cases}$$

Example

The specification of the power system gives rise to a NEDA with four EDAs, representing the main power component ($\mathfrak{A}_1$), the two batteries ($\mathfrak{A}_2$ and $\mathfrak{A}_3$), and the monitor ($\mathfrak{A}_4$). This yields the following mappings, assuming that P and B represent any global mode (m_1, m_2, m_3, m_4) with $m_1 = \texttt{primary}$ or $m_1 = \texttt{backup}$, respectively:

$$\begin{array}{rcl}
\alpha(P) & = & \{1,2,4\}; \\
\alpha(B) & = & \{1,3,4\}; \\
EC(P) & = & \{(2.\texttt{empty}, 1.\texttt{batt1}.\texttt{empty})\}; \\
EC(B) & = & \{(3.\texttt{empty}, 1.\texttt{batt2}.\texttt{empty})\}; \\
DD(P)(1.\texttt{alert}) & = & 4.\texttt{alert}, \\
DD(P)(2.\texttt{tryReset}) & = & 4.\texttt{alert}, \\
DD(P)(4.\texttt{voltage}) & = & 2.\texttt{voltage}, \\
DD(P)(4.\texttt{alert}) & = & (4.\texttt{voltage < 4.5}), \\
DD(B)(1.\texttt{alert}) & = & 4.\texttt{alert}, \\
DD(B)(3.\texttt{tryReset}) & = & 4.\texttt{alert}, \\
DD(B)(4.\texttt{voltage}) & = & 3.\texttt{voltage, and} \\
DD(B)(4.\texttt{alert}) & = & (4.\texttt{voltage < 4.5}).
\end{array}$$

This four-component NEDA of the power system exhibits the transitions shown in Figure 3.2. Configurations are represented as $c_1 \parallel c_2 \parallel c_3 \parallel c_4$ where c_1 is the current configuration of component Power, c_2 of batt1, c_3 of batt2, and c_4 of component mon. In each configuration, the mode information of active components is underlined.

3.5 Graphical Notation

As an extension of the COMPASS project, we, together with Ellidiss, developed a graphical notation of SLIM and a graphical editor for it. Given SLIM's component-oriented nature, it lends itself well for a visualisation as a graph-like shape. This enables the engineer to visually see port connections between components quickly. Furthermore, many SLIM syntax rules that are typically checked after construction of the textual SLIM model, can be checked for while it is being created through drag-and-drop of graphical elements. All this together increases the usability of the modelling language, and hence aids to its acceptance by engineers.

The graphical notation is described in [Nol11a] and it is inspired by AADL Annex A [SAE-AS5506/1], the graphical AADL notation. It is slightly adapted to handle SLIM specifics features, like flows and hybridity. It is extended with a novel notation for error models, which is not present in AADL Annex A. An example of the notation is shown in Figure 3.3.

$$
\begin{gathered}
\langle \underline{\text{primary}}, [\text{alert} \mapsto \bot] \rangle \parallel \\
\langle \underline{\text{charged}}, [\text{energy} \mapsto 1.0, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 6.0] \rangle \parallel \\
\langle \text{charged}, [\text{energy} \mapsto 1.0, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 6.0] \rangle \parallel \\
\langle \underline{m_0}, [\text{voltage} \mapsto 6.0, \text{alert} \mapsto \bot] \rangle \\
\Downarrow 30.0 \\
\langle \underline{\text{primary}}, [\text{alert} \mapsto \bot] \rangle \parallel \\
\langle \underline{\text{charged}}, [\text{energy} \mapsto 0.4, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 6.0] \rangle \parallel \\
\langle \text{charged}, [\text{energy} \mapsto 1.0, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 6.0] \rangle \parallel \\
\langle \underline{m_0}, [\text{voltage} \mapsto 6.0, \text{alert} \mapsto \bot] \rangle \\
\Downarrow \tau \\
\langle \underline{\text{primary}}, [\text{alert} \mapsto \bot] \rangle \parallel \\
\langle \underline{\text{charged}}, [\text{energy} \mapsto 0.4, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 4.8] \rangle \parallel \\
\langle \text{charged}, [\text{energy} \mapsto 1.0, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 6.0] \rangle \parallel \\
\langle \underline{m_0}, [\text{voltage} \mapsto 4.8, \text{alert} \mapsto \bot] \rangle \\
\Downarrow 10.0 \\
\langle \underline{\text{primary}}, [\text{alert} \mapsto \bot] \rangle \parallel \\
\langle \underline{\text{charged}}, [\text{energy} \mapsto 0.2, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 4.8] \rangle \parallel \\
\langle \text{charged}, [\text{energy} \mapsto 1.0, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 6.0] \rangle \parallel \\
\langle \underline{m_0}, [\text{voltage} \mapsto 4.8, \text{alert} \mapsto \bot] \rangle \\
\Downarrow \tau \\
\langle \underline{\text{primary}}, [\text{alert} \mapsto \top] \rangle \parallel \\
\langle \underline{\text{charged}}, [\text{energy} \mapsto 0.2, \texttt{tryReset} \mapsto \top, \text{voltage} \mapsto 4.4] \rangle \parallel \\
\langle \text{charged}, [\text{energy} \mapsto 1.0, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 6.0] \rangle \parallel \\
\langle \underline{m_0}, [\text{voltage} \mapsto 4.4, \text{alert} \mapsto \top] \rangle \\
\Downarrow \text{empty} \\
\langle \underline{\text{primary}}, [\text{alert} \mapsto \bot] \rangle \parallel \\
\langle \text{depleted}, [\text{energy} \mapsto 0.2, \texttt{tryReset} \mapsto \top, \text{voltage} \mapsto 4.4] \rangle \parallel \\
\langle \underline{\text{charged}}, [\mapsto 1.0, \texttt{tryReset} \mapsto \bot, \text{voltage} \mapsto 6.0] \rangle \parallel \\
\langle \underline{m_0}, [\text{voltage} \mapsto 6.0, \text{alert} \mapsto \bot] \rangle \\
\Downarrow \vdots
\end{gathered}
$$

Figure 3.2: An example trace of the Power system.

3.6 Differences between SLIM, AADL and Annexes

AADL 1.0 by itself expresses only architecture, and not so much behaviour. The behaviour in AADL 1.0 is primary emergent by subprograms that are invokable within the architecture. Alternatively, the annex mechanism can be used to introduce behaviour using the specification language as described in AADL's Behaviour Annex 1.0 [SAE-AS5506/2]. In SLIM, architecture and behaviour are intimately

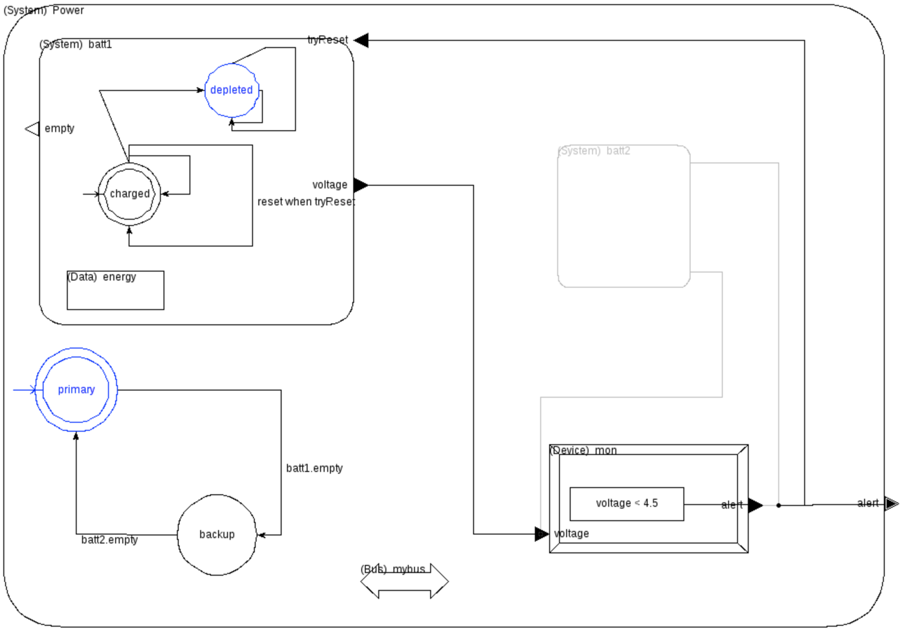

(System) Power
(System) batt1
tryReset
depleted
empty
charged
voltage
reset when tryReset
(Data) energy
(System) batt2
primary
batt1.empty
(Device) mon
voltage < 4.5
alert
alert
voltage
backup
batt2.empty
(Bus) mybus

Figure 3.3: Power system visualised using the SLIM graphical notation.

integrated, incorporating aspects from AADL 1.0, its Error Annex 1.0 and its Behaviour Annex 1.0 in a coherent and integrated way. Furthermore, SLIM is designed to be expressive enough for the space systems engineering domain, while on the other hand formal enough to be subjected to formal analysis. This implies that SLIM aims to suit late requirements engineering and the preliminary design phase, whereas pure AADL 1.0 typically is employed in the detailed design phase with possible refinements of the model to the early implementation phase.

SLIM – AADL 1.0 The different intended usage scenario's of SLIM and AADL 1.0 have led to small deviations between the two languages. From a syntactic perspective, SLIM differs from AADL 1.0 on the following:

- Subprograms. Not present in SLIM, present in AADL. SLIM does not link at source level with other languages.
- Properties. Not present in SLIM, present in AADL. In SLIM all required characteristics are part of the core language.
- Annexes. Not present in SLIM, present in AADL. SLIM embeds aspects of relevant annexes as part of the language's syntax.
- Flows. In SLIM flows are concrete data paths expressed by flow expressions. In AADL flows are abstract data paths expressed by flow paths.

- Data. In SLIM `data` declarations are always variables. In AADL they can be a structure of variables. Furthermore, in SLIM data elements need to have an initial value. In AADL, this is undefined.
- Ports. In SLIM, a port is either an `event` or `data` port. In AADL, there are also `event data` ports and port groups.
- Event queuing. In SLIM, transitions rendez-vous on events. In AADL, events can be queued on the event data ports.

SLIM – Error Annex 1.0 A majority of the elements from AADL's Error Model Annex have been incorporated in SLIM. The approach in SLIM is however simplified, and more oriented towards data-failures rather than event-failures. The resulting differences are:

- Interaction between error and nominal model. In SLIM, model extension using fault injections are used. With the Error Model Annex, the annex keyword is used to associate an error component with a nominal component. Furthermore, in SLIM, the error model influences the nominal model through errors in data elements. In the Error Model Annex, the error model influences the nominal model through mangling of events.
- Error propagation. In SLIM, propagated errors are matched by their identifier. In the Error Model Annex, filters, guards and masks can be defined for expressing fine-grained propagations.
- Derived error models. In SLIM, only basic error models are possible. In the Error Model Annex, both basic and derived error models are possible. The latter models describe behaviour of a component as a function of the error states of its subcomponents.

SLIM – Behaviour Annex The Behaviour Annex was still under development while SLIM was designed. SLIM's behavioural specification is therefore inherently different than that in the Behaviour Annex. In SLIM, nominal behaviour emerges from transitions between modes. In addition to them being event triggered, as in AADL, in SLIM they can also be guarded by boolean expressions over data elements and can change values of data elements. With the Behaviour Annex, a state machine that allows guards and effects is defined that co-exists with the mode transition system. Another substantial difference is that SLIM allows for the expression of hybrid evolution of continuous elements using trajectory equations and mode invariants. These are neither present in AADL nor in its Behaviour Annex.

To conclude, the major difference between SLIM, AADL and its annexes, is that SLIM provides a coherent, and complete formal semantics for the combination of dynamic reconfigurable architectures, their functional behaviour as mode transitions and errors to that behaviour due to error models. For AADL, there are semantics for AADL in combination with the Error Annex, or AADL in combination with the Behaviour Annex, but not the combination of all three. This makes SLIM suitable for capturing dynamic reconfigurable system architecture *and* behaviour *under* degraded conditions. It is also expressible with AADL and its annexes, but the formal semantics of the combination is undetermined. SLIM's formal semantics are defined with the rigour of transition systems. It leaves no unambiguity about concurrent behaviour, along with its typical issues of atomicity and non-determinism.

3.7 Discussion

AADL 1.0 without its annexes provides the means to express dynamic reconfigurable embedded systems architectures from both the hardware and software perspective, supporting the system-software engineering process. From this perspective, architecture is the glue and the foundation upon which the system is built. The first tools for AADL have therefore been designed for this aim. An example of this is TASTE [Per+12], a toolset developed by the European Space Agency with its partners from the space industry. It allows engineers to integrate heterogeneous source code (e.g. in C, ADA) and generate an executable prototype of the system. During our research visit at ESA we shared the corridor with the principal developers of TASTE. Our discussions and experiments resulted in most of Section 3.6.

AADL's properties mechanism provides means to express timing characteristics that impact schedulability of tasks by accounting for their worst-case execution times. An approach towards schedulability analysis in AADL is described in [Sin+05] and is called Cheddar. It has become part of TASTE. Afterwards, the first wave of verification tools were developed that checked for functional correctness. Our work was one of the first. Around the same time, a work was published [RKH09] that maps the time-constrained event-behaviour captured by AADL's mode transitions to timed Petri nets. They also mention the possibility of using colored Petri nets to handle data-dependent constraints. When the Behaviour Annex started to mature, many tool-supported approaches emerged that tackled verification of behaviour. One of the earliest maps AADL and a fraction of its Behaviour Annex to BIP [Chk+08]. BIP is then mapped to the IF model checker [Boz+04], enabling the verification of safety properties. Another work [Ber+09] maps AADL and a fraction of its Behaviour Annex to Fiacre, which is then mapped to a timed Petri net. Later, after the acceptance of the Behaviour Annex, a work [Bjo+11] was published that describes a denotational semantics for a subset of AADL and its Behaviour Annex. A CTL model checker called ABV was built that implemented that semantics. All these works follow the asynchronous nature of AADL's component interaction. A work that reinterprets this interaction as a synchronous system is described in [Bae+11]. It maps a fraction of AADL and its Behaviour Annex to real-time Maude. Using the Maude model checker, LTL and timed CTL properties can be verified. All these approaches follow the level of abstraction aimed originally for AADL, where hardware components induce a great part of the system architecture and software induces behaviour subject to analysis. In our approach, hardware and software are considered perceived from a higher abstraction-level, namely to the level where they have a behaviour that provides a functionality. As such, in SLIM, they are seen as communicating components, and the hierarchy of components induces the system architecture, not only the hardware. Second, all these approaches implicitly or explicitly have a formal semantics defined, but neither of them account for the probabilistic aspect of error behaviour. Neither do they cover hybrid behaviour. In their current shape, their formal semantics are designed to cover a specific analysis, like model checking, but are not suitable for a wide range of analyses (including probabilistic risk assessments for example), as we aimed for with SLIM. We argue that a formal semantics needs to cover a broad range of behavioural aspects in order to ensure that analyses provide results that are coherent with each other. For this, a formal semantics needs not only to cover discrete behaviour, but also timing, probabilistic and hybrid behaviour, as SLIM does.

We had few approaches in mind for interrelating nominal behaviour with error behaviour. In AADL and its Error Annex, the annex mechanism is used to attach erroneous behaviour to a nominal component. Then rules are defined that allow erroneous events to mangle nominal events. In [RKK07], it is shown how to map the result to a generalised stochastic Petri net. We considered this approach too, but found that the mangling rules ill-suitable to specify data failures. The other option was to have the user directly specify the extended model. This however defies the separation of concerns of the nominal and error behaviour, which are traditionally served by different fields of engineering, namely system engineering versus RAMS engineering. Model extension provides for the separation of concerns while enabling injection of data-failures. It is adapted from FSAP's approach towards model extension [BV03a]. Both FSAP and our approach separate the concerns of nominal behaviour and the fault model, which are then married through fault injections. The slight difference is that FSAP did not support arbitrary error transition systems, but only ones shaped as a two-state loop, where one state represents a nominal operating mode, and the other one a particular failure mode. This modification is necessary to express multiple degraded modes of operation.

Our early approach towards fault injection differs from our final approach. Initially, we defined fault injections between an error model implementation and a nominal model *implementation* instead of a nominal *instance*. During our early evaluation, we observed that the former approach was too limited, because it associated a single error model with the same set of fault injections to all instances of the affected component implementation. It would not be possible to associate different erroneous behaviour to a component implementation with the same abstract behaviour, which for example can happen if a particular component design is realised with a different choice of materials.

Our approach of combining the nominal and error model through model extension also inspired our users to identify their own needs. Throughout our case studies, of which the one in Chapter 6 is an example, modellers have expressed the wish to inject faults into trajectory equations. They could for example be overridden by a faulty trajectory equation. Furthermore, it some cases, it was desired to force a nominal event to happen upon entering an error state. The principal effect is similar to the approach by AADL and the Error Annex, which only support this kind of errors. Lastly, we observed the possible need to make fault injections, and perhaps even the associated error model implementation, additionally mode-dependent. This enables engineers to express different error behaviours during different nominal modes of operation. For example, the erroneous behaviour of a satellite might briefly alter during launch and a solar eclipse. The exact implications and the desirability to express such granular erroneous behaviour needs to be further investigated.

4

Formal Architecture Analysis

Verification and validation in early phases of the space systems engineering process, like phase A and B, is typically oriented towards inspection, review and analysis, as testing requires are particular design fidelity that is not yet attained in those phases. In this dissertation, and this chapter in particular, analysis is emphasised, as we define several formal analysis methods over our modelling formalism described in Chapter 3. They are based on algorithms from state-of-the-art formal methods research. The results they automatically generate are in the current engineering practice obtained manually, or are a formal expression of reasoning that occurs implicitly during verification and validation of the design in phase B.

The offered analyses are categorised into the following: correctness analysis, safety and dependability analysis, fault management effectiveness analysis, probabilistic risk assessment and performability analysis. Each section represents an analysis, whose purpose is described, along with the approach towards its implementation, an example, and a discussion on its strengths and known limitations. The referred analyses are scoped to the capabilities of the COMPASS toolset (cf. Chapter 5). The examples are based on the sensor-filter model which is provided in Appendix A. It is possible to define other analyses over SLIM models, using it formal semantics, like criticality analysis [Ern12]. These are however out of the scope of this dissertation.

4.1 Fault Injection

Fault injection marries fields of engineering which are traditionally separated. These are all the engineering disciplines that concern themselves with the nominal behaviour of the system, as this is the intended and typically the behaviour the system exhibits most of the time. The other one is safety and dependability engineering, whose engineers concern themselves with behaviour that is typically sporadic, yet have potential critical consequences. These disciplines generate their own artefacts, which through fault injections are combined into an extended model (cf. Section 3.3) that exhibits both nominal and non-nominal behaviour.

Fault injections are orthogonal upon the analyses described in subsequent sections. Its main purpose is to apply the effects of faults upon the nominal behaviour, which typically leads to non-nominal behaviour. Different configurations of fault injections trigger different non-nominal behaviours, which might require understanding depending on their anticipated criticality.

In our context, the error models and fault injections are developed by the safety and dependability engineers using the early FMECA tables and fault trees as input. This is a modelling process in itself, which may reveal improvements and additions that can be used to enhance the FMECA tables. If neither FMECA tables and fault trees are existent, typically boundary conditions need to be analysed of data elements, and reason how these boundary conditions lead to failures. The outcome of such an analysis indicates for possible fault injections. The resulting set of fault injections can be used to generate the FMECA tables (cf. Section 4.8) and fault trees (cf. Section 4.5), which can be provided as input to the space system engineering process.

4.1.1 Approach

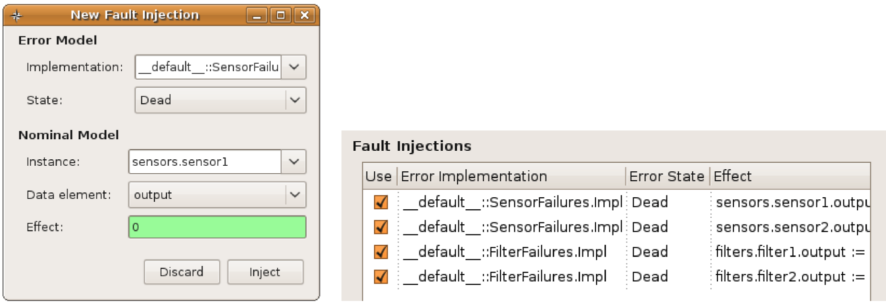
After loading the nominal model and the error model, it becomes possible to relate them by fault injections. A single fault injection consists of the following parts:

- reference to an error model implementation,
- reference to an error state occurring in this error model implementation,
- reference to the affected nominal component instance,
- reference to the affected data element,
- a fault expression, representing the effect of the fault.

These are the elements for performing model extension, as described in Section 3.3.

4.1.2 Complexity Analysis

Fault injection is not time nor memory consuming by itself. It does affect the model extension process, which takes the fault injections, the nominal model and the error model as inputs. Model extension takes $O(n)$ time where n is the amount of fault injections. Yet, the absolute time spent on model extension is neglectible compared to the analyses described in this chapter, as the number of fault injections is typically small and the analyses have a heavier time-complexity. A particular choice of fault injections however affects the resulting extended model. If the injected faults enable elaborate (recovery or degraded) behaviour, the state space associated with the extended model increases significantly. This was also observed



Error Model

Implementation: __default__::SensorFailu

State: Dead

Nominal Model

Instance: sensors.sensor1

Data element: output

Effect: 0

Fault Injections

Use	Error Implementation	Error State	Effect
☑	__default__::SensorFailures.Impl	Dead	sensors.sensor1.outpu
☑	__default__::SensorFailures.Impl	Dead	sensors.sensor2.outpu
☑	__default__::FilterFailures.Impl	Dead	filters.filter1.output :=
☑	__default__::FilterFailures.Impl	Dead	filters.filter2.output :=

(a) Form for entering a single fault injection.

(b) Overview of injected faults.

Figure 4.1: Fault injection in COMPASS.

during our case studies (cf. Chapter 6). We observed that fault injections increase the amount of system behaviours, and hence its state space. The degree does not necessarily correlate with the amount of fault injections, but rather with the kind of fault injections. This observation is discussed in depth in Section 6.5.

4.1.3 Example

An example of a fault injection in COMPASS on the sensor-filter case study (cf. Appendix A) is shown in Figure 4.1(a). It describes that whenever the sensor dies (i.e. error state Dead in error model SensorFailures), the first sensor's output (i.e. port output of sensors.sensor1) sticks to 0. Other possible fault injections for the sensor-filter case study are:

Error model		Nominal model		
Implementation	State	Instance	Data element	Effect
SensorFailures	Dead	sensors.sensor2	output	0
FilterFailures	Dead	filters.filter1	output	15
FilterFailures	Dead	filters.filter2	output	15

Once a fault is injected, it automatically becomes an enabled injection. This can be verified by inspecting the checked box on the left in Figure 4.1(b), which shows the list of fault injections. By enabling/disabling an injection, different fault configurations can be explored and understood. A particular combination of fault injections upon a model gives rise to a particular extended model, and this can be inspected by viewing it in textual SLIM form (cf. Section 3.3).

4.1.4 Discussion

The theoretically oriented discussion on fault injection coincides with that of model extension, and as such is already part of Section 3.3. On a usability note, we received feedback from engineers that our approach does not require that *all* error states have faults injected. It can happen that an error state is entered, but no faults effects are observed. In industrial practice, this is rather uncommon, since error states typically have effects. We intentionally kept our original approach, enabling the modeller to run analyses without the need for injecting all faults, which is a typical use case for a model under construction.

4.2 Properties

The premise of an analysis is the hypothesis of a system's behaviour, which needs to be proven or rejected. With respect of analysing a SLIM model, the formal shape of a hypothesis is dependent on the employed analysis. For most analyses, the hypothesis is shaped as a property. In case of model verification, the model is typically derived from design documentation, and the model needs to be checked with the requirements. Each requirement is then represented by properties. Typically, detailed and specific requirements are mapped to a single property, whereas more general requirements can be traced to multiple properties as they are aimed to cover more of the system's behaviour. For model validation, the model and properties are typically not obtained from the design and the requirements. The model is typically derived from the requirements and parts of the early design and then needs to be checked with the stakeholder's understanding. This means that properties are coming from the user or stakeholder directly. This usually happens during requirements engineering and early design, where both requirements and design are not in their definitive stages, and formal analysis aids to their crystallisation. For both verification and validation, property is a characteristic of the model that we check for. They are cornerstone to analysis, and for most of them, they are, together with the model, input to the analysis. Since the formal shape of a hypothesis depends on the employed analysis, its use varies across different analyses.

For formal analysis, properties need to be expressed formally, typically according to a logic, like linear temporal logic (i.e. LTL) or computational tree logic (i.e. CTL). These logics have however a steep learning curve to those unfamiliar with notations from mathematical logic. Exposing them in their pure form, as employed by academics, is likely to be disadvantageous to its acceptance into an engineering process, where steep learning curves require a clearly quantified return on investment. In our approach, we implemented a middle-way, which balances the need for formality, while keeping its format closer to engineering practice.

4.2.1 Approach

Our approach is inspired by Dwyer et. al [DAC99], who developed a pattern-based approach towards requirements specification. Their study, based on analysing over 500 industrial requirements, revealed that 92% of them can be captured in eight patterns. A pattern contains blank parameters, which have to be filled, i.e. instantiating the pattern to a property. The pattern has a single interpretation, but different notational expression, depending on the user's familiarity. It can be noted in structured (English) prose, or graphically in Graphical Interval Logic [Dil+94], or as LTL or CTL formula. Due to these direct translations, a user unfamiliar with LTL can prefer the structured prose instead. In our approach, we distinguish three classes of patterns, namely:

Propositional	These properties correspond to a proposition, like `error` = `error:transmittedFault` and are used for particular analysis like fault tree analysis or failure modes and effects analysis.
Functional	For performing qualitative analyses (that is, without any probabilistic aspect), the properties should be expressed using CTL/LTL patterns as given in Table 4.2.
Probabilistic	For performing quantitative analyses that are probabilistic in nature. They are outlined in Table 4.3.

The patterns have parameters to be filled up by the engineer. They are composed of operators that connect atomic propositions, which may be the following:

- Ports, e.g. `sc.ssc.port1`, where `sc` is a subcomponent identifier, `ssc` is a subcomponent of the implementation of `sc` and `port1` is the port identifier.
- Data subcomponents, e.g. `sc.data1`, where `sc` is a subcomponent identifier and `data1` is the data subcomponent identifier.
- Constant values:
    - Integers, e.g. `42`,
    - Reals, e.g. `42.001`,
    - Booleans, e.g. **`true`**,
    - Enumeration literals, e.g. **`enum`**`:C1`, where `C1` is the literal.
- Mode variables, e.g. `sc.`**`mode`** where `sc` is the identifier of a subcomponent of the root component, representing the current mode of the subcomponent `sc`.
- Mode names, e.g. **`mode`**`:m1`, where `m1` is the mode.
- Error variables, e.g. **`error`**, referring to the current state of the error model that has been associated with the root component.

Table 4.2: The functional patterns for capturing properties.

Pattern	Description	Logic (CTL/LTL)
Global absence	The atomic proposition $\boxed{\phi}$ never holds.	$\forall\Box\neg\phi$ $\Box\neg\phi$
Global existence	The atomic proposition $\boxed{\phi}$ shall eventually hold.	$\forall\Diamond\phi$ $\Diamond\phi$
Global universal	The atomic proposition $\boxed{\phi}$ globally holds.	$\forall\Box\phi$ $\Box\phi$
Global precedence	The atomic proposition $\boxed{\phi}$ globally precedes $\boxed{\psi}$.	$\neg\exists(\neg\phi \bigcup(\psi \wedge \neg\phi))$ $(\Box\neg\psi) \vee (\neg\phi \bigcup \psi)$
Global response	Whenever the atomic proposition $\boxed{\phi}$ holds, this is eventually responded with $\boxed{\psi}$.	$\forall\Box(\phi \implies \forall\Diamond\psi)$ $(\Box\phi \implies \Diamond\psi)$
Exists response[a]	Whenever atomic proposition $\boxed{\phi}$ holds, it may be eventually responded with $\boxed{\psi}$.	$\forall\Box(\phi \implies \exists\Diamond\psi)$

[a] This is a CTL-only pattern. It is not available for activities that require LTL patterns.

Table 4.3: The probabilistic patterns. p is the probability of interest.

Pattern	Description	Logic (CSL)
Probabilistic invariance	The invariant $\boxed{\phi}$ holds continuously between $\boxed{t_1}$ and $\boxed{t_2}$ with probability p.	$\mathscr{P}_{=p}(\Box^{[t_1,t_2]}\phi)$
Probabilistic existence	$\boxed{\phi}$ will eventually become true within $\boxed{t_1}$ and $\boxed{t_2}$ with a probability p.	$\mathscr{P}_{=p}(\Diamond^{[t_1,t_2]}\phi)$
Probabilistic until	$\boxed{\phi}$ will eventually become true within $\boxed{t_1}$ and $\boxed{t_2}$ after $\boxed{\psi}$ held continuously with a probability p.	$\mathscr{P}_{=p}(\psi \bigcup^{[t_1,t_2]} \phi)$
Probabilistic precedence	$\boxed{\phi}$ precedes or enables $\boxed{\psi}$ within $\boxed{t_1}$ and $\boxed{t_2}$ with a probability p	$\mathscr{P}_{=1-p}(\neg\phi \bigcup^{[t_1,t_2]}(\neg\phi \wedge \psi))$
Probabilistic response	After $\boxed{\phi}$ holds, $\boxed{\psi}$ must become true within $\boxed{t_1}$ and $\boxed{t_2}$ with a probability p.	$\mathscr{P}_{=1}\Box(\phi \implies \mathscr{P}_{=p}(\Diamond^{[t_1,t_2]}\psi))$

- Error state names, e.g. `error:e1`, where `e1` is the error state.

4.2.2 Complexity Analysis

The translation of a pattern to its logical form is one-to-one. The complexity of the translation process is therefore constant time. In the tool, there is additionally a check on the validity of the atomic propositions by analysing the model under analysis. This check, which computes all valid atomic propositions and their corresponding types is linear to the size of the model. The absolute time needed for both the translation and the check is neglectible compared to the analyses in this chapter.

4.2.3 Example

Properties that can be used for the sensor-filter case study (cf. Appendix A) are:

- Propositional `value >= 15`, i.e. that the sensor fails.
- Propositional `value = 0 or value >= 15`, i.e. that either the sensor or filter fails.
- Global response that the monitor reacts on filter failures, i.e.

$$\begin{aligned}\phi &\equiv \text{value} = 0\ \mathbf{and}\ \text{filters.}\mathbf{mode} = \mathbf{mode}\text{:Backup}\\ \psi &\equiv \text{alarmF}\end{aligned}$$

- Probabilistic existence that the sensors or filters die within 76 time units, i.e.

$$\begin{aligned}\phi \equiv\ & (\text{sensors.sensor1.}\mathbf{error} = \mathbf{error}\text{:Dead}\ \mathbf{and}\\ & \ \text{sensors.sensor2.}\mathbf{error} = \mathbf{error}\text{:Dead})\ \mathbf{or}\\ & (\text{filters.filter1.}\mathbf{error} = \mathbf{error}\text{:Dead}\ \mathbf{and}\\ & \ \text{filters.filter2.}\mathbf{error} = \mathbf{error}\text{:Dead})\\ t_1 \equiv\ & 0\\ t_2 \equiv\ & 76\end{aligned}$$

- Probabilistic precedence that the sensor bank fails before the filter bank fails within 512 time units, i.e.

$$\begin{aligned}\phi &\equiv \text{sensors.sensor2.}\mathbf{error} = \mathbf{error}\text{:Dead}\\ \psi &\equiv \text{filters.filter2.}\mathbf{error} = \mathbf{error}\text{:Dead}\\ t_1 &\equiv 0\\ t_2 &\equiv 512\end{aligned}$$

4.2.4 Discussion

Patterns have been used in many areas as a solution to reoccurring problems. For software specifically, software design patterns have been thoroughly investigated [Gam+95] and applied to software engineering practice. The first work we are aware of that introduced the concepts of patterns to formal requirements is that by Dwyer et. al [DAC99]. Our approach can be considered as an implementation of that work. We added an additional pattern, the existential response, to cover our own anticipated need for checking whether the system has a possibility for repair. The work by Dwyer et. al led to several derivative works [Kon+03; Smi+02], among which the probabilistic one by Grunske [Gru08] is used by us as well.

Regardless of the pattern system used, the choice of supported atomic propositions correlates directly to the modelling language. In our case, data elements and modes can be referred to. In our case studies (cf. Chapter 6), this has shown to be sufficient. Shortly after development of the COMPASS toolset, an idea was raised to support events as atomic propositions as well. An issue with that idea that needs to be resolved is whether a referred event in an atomic proposition relates to enabledness of the transition, or whether it refers to a state that directly follows after the occurrence of the event. Also, its exact use cases need to be further investigated.

4.3 Simulation

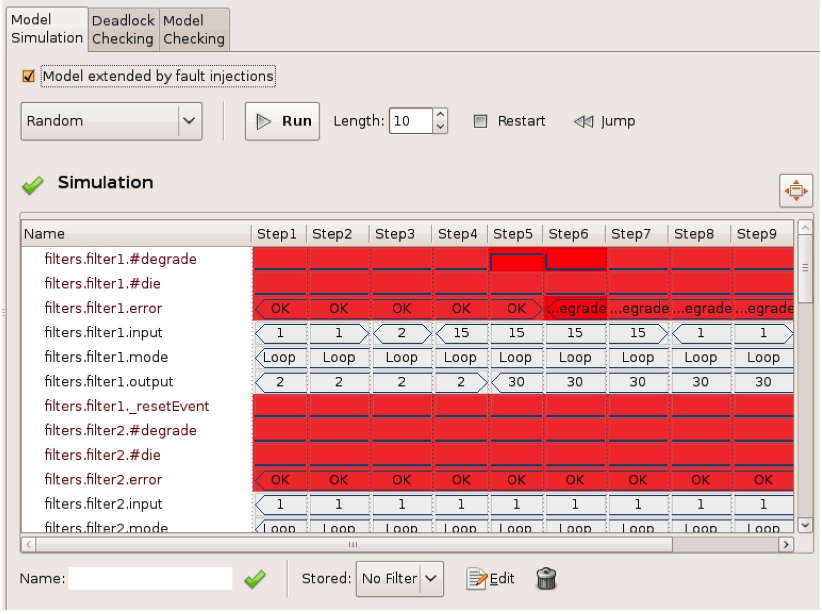
A simulation produces a trace of states occurring in the order from the initial state. It is one of the most natural types of analysis. Simulation (cf. Figure 4.2) is typically used as a sanity check before more resource-consumptive analyses are run. In our approach, we defined three ways of generating a simulation:

- randomly, where at each state, a successor state is randomly chosen from the one-step reachable states,
- step-by-step, where at each state, a successor state is chosen by the user from the one-step reachable states,
- constraint-based, where at each state, a successor state is determined by user-defined constraints on the one-step reachable states.

Simulation can be performed with fault injections or without, allowing for exploring purely nominal behaviour or the extended behaviour.

4.4 Model Checking

Whereas simulation explores a single trace, model checking exhaustively explores all traces. If fault injections are enabled, these traces also account for behaviours



☑ Model extended by fault injections

Random

Length: 10

Name	Step1	Step2	Step3	Step4	Step5	Step6	Step7	Step8	Step9
filters.filter1.#degrade									
filters.filter1.#die									
filters.filter1.error	OK	OK	OK	OK	OK	...egrade	...egrade	...egrade	...egrade
filters.filter1.input	1	1	2	15	15	15	15	1	1
filters.filter1.mode	Loop	Loop	Loop	Loop	Loop	Loop	Loop	Loop	Loop
filters.filter1.output	2	2	2	2	30	30	30	30	30
filters.filter1._resetEvent									
filters.filter2.#degrade									
filters.filter2.#die									
filters.filter2.error	OK	OK	OK	OK	OK	OK	OK	OK	OK
filters.filter2.input	1	1	1	1	1	1	1	1	1
filters.filter2.mode	Loop	Loop	Loop	Loop	Loop	Loop	Loop	Loop	Loop

Name:

Stored: No Filter

Figure 4.2: Trace of a simulation from the COMPASS toolset.

due to fault effects. In that respect model checking resembles a form of exhaustive use case testing (cf. Section 2.1.4). Model checking is however precise and systematic. It avoids overlooking use cases that lead to difficult to spot boundary conditions (cf. boundary analysis in Section 2.2.4). The set of all traces can be visualised as state space. For realistic models, the state space easily grows into an order of trillions states, making them difficult to visualise in a sensible manner for manual inspection. Hence, instead of visualising the state space, we use it to check a property (cf. Section 4.2) against it. If the property holds for the state space, it is a property that holds for the model. Otherwise, the model checker can provide a counterexample describing a trace that violates the property. Model checking is typically used to verify the functional correctness, in which case properties represent desired global behaviour.

4.4.1 Approach

Model checking is formally defined as $M \models \phi$, where M is the model and ϕ is the property to be verified. The model M is our case expressed in SLIM, i.e. M_{SLIM}. The property ϕ is expressed in a property pattern (cf. Section 4.2) which has a direct CTL equivalent. Instead of model checking the model in SLIM directly, we follow a translation based approach. Using SLIM's formal semantics (cf. Section 3.4), we map M_{SLIM} to NuSMV's input language, i.e., M_{SMV}. This way, we use NuSMV's existing model checking algorithms to model check SLIM. In case a counterexample is returned by NuSMV, all NuSMV symbols are remapped back to their SLIM counterparts.

We distinguish and two techniques for checking $M \models \phi$, namely one using binary decision diagrams (BDD) as data-structures for representing the state space, and one using SAT-formula's to represent (a part of) the state space. Both approaches have their advantages and disadvantages.

The BDD-based technique is intrinsically limited to finite structures, and hence to the finite part of SLIM, i.e. models containing `real` variables and trajectory equations are not suitable for this approach. Early phase B design information typically does not describe system behaviours in such detail, and rather expresses a finite abstraction of this, which are perfectly modelled using the finite part of SLIM.

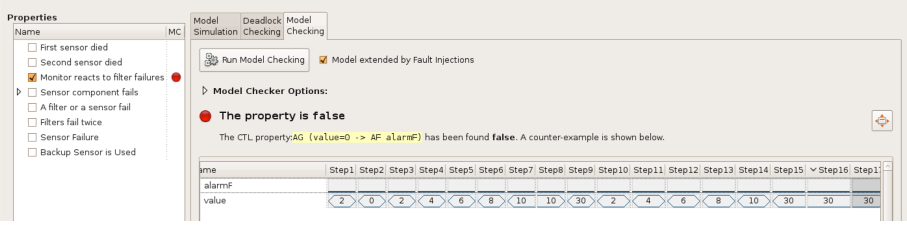
The SAT-based technique handles both finite and infinite structures, and hence covers SLIM's full language. It encodes all possible traces up to a given bound, and checks whether a property holds. If it does, it is ensured that the property holds up to the depth given by the bound. For behaviours beyond that bound, the correctness is inconclusive. If a violation is detected, and a counterexample is generated, is it ensured that it is a correct counterexample. For this, the SAT-based approach is typically suitable for falsification of properties, especially if the user suspects that the violation of the property occurs early in the state space.

4.4.2 Complexity Analysis

The complexity for translating a SLIM model to NuSMV takes linear time in the size of the model. This is neglectible compared to the complexity of model checking. BDD-based model checking uses the CTL representation of the properties since all the patterns in Section 4.2 have a mapping to CTL. CTL model checking has a complexity in $O(|\mathrm{TS}| \cdot |\phi|)$ where $|\mathrm{TS}|$ is the size of the state space and $|\phi|$ the size of the formula [BK08]. For the SAT-based techniques, this is different. It is based on [Bie+99], which encodes a LTL model checking into a Boolean satisfiability problem. LTL model checking is known to be PSPACE-complete [BK08]. Boolean satisfiability is however known to be NP-complete, meaning that the SAT-based technique of LTL model checking is NP-complete as well [Bie+99].

4.4.3 Example

From the properties in Section 4.2.3, the global response property is amenable for model checking the sensor-filter model (cf. Appendix A). It states that whenever $\mathtt{value} = 0$ **and** $\mathtt{filters}.\mathbf{mode} = \mathbf{mode}{:}\mathtt{Backup}$, this is eventually responded by $\mathtt{alarmF}$. As the sensor-filter model is a finite model, BDD-based techniques are used by default, although SAT-based techniques can be chosen instead. The property as it is holds on the model. If we however change ϕ to $\mathtt{value} = 0$, and model check the resulting property against the sensor-filter model, then we get a



- ☐ First sensor died
- ☐ Second sensor died
- ☑ Monitor reacts to filter failures ●
- ☐ Sensor component fails
- ☐ A filter or a sensor fail
- ☐ Filters fail twice
- ☐ Sensor Failure
- ☐ Backup Sensor is Used

☑ Model extended by Fault Injections

● **The property is false**

The CTL property:`AG (value=0 -> AF alarmF)` has been found **false**. A counter-example is shown below.

ame	Step1	Step2	Step3	Step4	Step5	Step6	Step7	Step8	Step9	Step10	Step11	Step12	Step13	Step14	Step15	Step16	Step1
alarmF																	
value	2	0	2	4	6	8	10	10	30	2	4	6	8	10	30	30	30

Figure 4.3: A model checking counterexample whether the filter alarms are raised when `value` drops to 0.

counterexample (cf. Figure 4.3). The counterexample shows that at the second state (step 2), `value` drops to 0. This is however not responded by `alarmF`, as the counterexample shows a loop (indicated by an downwards arrow) between the states in step 16 and step 17 where `alarmF` stays **false**.

4.4.4 Discussion

During one of our own internal side-track projects [Ode10], we occasionally encountered an inconsistency between outcomes of models checked with the SAT-based technique and the BDD-based technique. After inquiry with FBK, we were informed that this is related to deadlocks and the implementation of the model checking algorithms. When the BDD-based algorithms are used, the exploration of the state space is restricted to infinite paths, disregarding paths that lead to a deadlock. If a property is only violated on paths leading to a deadlock, NuSMV would not detect it, and it reports that the property holds for the model. When SAT-based algorithms are used, the exploration of the state space also regards paths leading to deadlocks. If such a path is violating the property, the model checker would report this as a counterexample. It can hence happen that the BDD-based algorithms report that a property holds, whereas the SAT-based algorithms do not, while checking the same model. The assumption of NuSMV is however that the model is free of deadlocks, and when this is the case, the model checking results are consistent. This reason is additional to the traditional scope of deadlock checking, namely whether the reactive system does not terminate. This behaviour is also a FAQ-point on NuSMV's FAQ page at `http://nusmv.fbk.eu/faq.html`.

Given the choice between the two techniques for model checking, SAT or BDD-based, there are three main concerns of preferring one over another. The first concern may be that of computing resources. Depending on the model, the BDD representing its state space may grow in unpredictable ways. This growth depends heavily on the chosen variable ordering. With the SAT-based technique, the SAT formula grows too, but in a more predictable manner. It has been shown earlier [Bie+99] that SAT-based techniques can be more efficient than BDD-based techniques on particular models. The second concern is that of expressiveness.

The BDD-based technique accepts only finite SLIM models, whereas the SAT-based technique accepts the full scope of SLIM, including the parts induced by trajectory equations and the use of real variables. The third concern is that of completeness. BDD-based techniques provide conclusive results, whereas SAT-based techniques by default remain inconclusive if the property is not falsified. An alternative SAT-approach, which is included in our work, is the Simple Bounded Model Checking technique. It uses a different encoding of the model checking problem to SAT, and can decide whether the bound is sufficient for completeness. This however only works for the finite part of SLIM, which is generally better served by the BDD-based technique .

When SAT-based techniques are used for checking hybrid SLIM models, three intertwining modelling issues needed to be accounted for, otherwise the checking may at first sight deliver unexpected outcomes. The first are Zeno cycles. The notion of “Zeno behavior” refers to traces involving an infinite number of discrete (i.e., mode transition) steps within a bounded period of time. An example of this is shown in Listing 4.1 in case `a = c = 0`. A cycle between `m0` and `m1` can be taken infinitely often within a bounded period of time. In the example, it even occurs without time passing.

Listing 4.1: Example exhibiting a Zeno cycle, a time lock and time divergence.

```
system implementation Timed.Imp
  subcomponents
    t0: data clock;
    t1: data clock;
  modes
    m0: initial mode while t0 <= b;
    m1: mode while t1 <= d;
  transitions
    m0 -[when t0 >= a then t1 := 0]-> m1;
    m1 -[when t1 >= c then t0 := 0]-> m0;
end Timed.Imp;
```

Zeno cycles contradict the natural assumption that only finitely many events can happen in a finite amount of time, which is valid for any realistic system. They occur due to abstraction, where the timed behaviour of the actions occurring in the Zero cycle are underspecified. Zeno cycles can be difficult to spot. They can be detected syntactically by checking whether the model meets the Strong Non-Zenoness property [Tri99]. In SLIM, one can check for this by checking every component of the system and on *each* cycle in the mode transition diagram of that component, there is a clock variable `t` that is reset, and that occurs in a transition guard of the form `t` > *k* or `t` >= *k*, where k is a strictly positive constant. A model that has the Strong Non-Zenoness property is ensured to be free from Zeno cycles. The property is sufficient but not necessary: there exist non-Zeno models which do not meet the Strong Non-Zeno property. Weaker forms of the property have been investigated which take additional factors into account, like synchronisation between components. In [BG06], a sufficient-and-necessary condition is proposed by a combination of static analysis and reachability analysis. It is proven to be correct on a subclass of models expressible in SLIM, namely only those with `clocks` (thus not **`continuous`** variables), resets are zero-valued and that mode invariants are right-closed intervals. In [HS11], this is further investigated and shown that the time-complexity for detecting Zeno cycles depends on the coarseness of abstraction employed for representing the state space. It is not (yet) known how to perform a precise Zeno cycle detection on more expressive (i.e. hybrid) models.

An additional potential issue with SAT-based techniques is that of deadlocks. A deadlock occurs in a state when for that state there is no transition possible, either by contradicting restrictions by the mode triggers and guards, or by the mode invariants. In Listing 4.1, a deadlock occurs when `a` = `2` and `b` = `1`. This is the simplest case. In more complicated settings, deadlocks can emerge from improper synchronisation between several components. The main problem with deadlocks is that the corresponding manual detection of them is non-compositional: the combination of two (or more) subsystems without deadlocks can result in deadlocking behaviour. Many forms of reachability analysis include the possibility of detecting deadlocks. For the purely real-timed case especially, the region-graph construction by Alur et al. [ACD93] can be used to detect sink nodes, which correspond to deadlocked states. Subsequent investigations have been investigated on more different state space representations, like zone-based representations. The work by [Tri99] from 1999 provides an overview on those. In a recent work by [HSW12], a precise characterisation of the coarsest abstraction has been determined while still preserving the correctness of reachability (and hence also deadlock checking). Deadlock detection techniques for more generalised hybrid systems are under active research. The work by [Aba+09] hooks into that area.

The third issue with SAT-based techniques is that of time divergence. This happens on infinite paths where time is allowed to grow arbitrarily. In Listing 4.1 a time divergent path occurs when the transition effect `t1 := 0` and the mode invariant `t1 <= d` are removed. Time divergence can easily be excluded by requiring for each component of the system that on every cycle in its mode transition system, every clock is reset at least once. This can be checked syntactically. As far as we know, there are no sufficient-and-precise conditions and algorithms for detection of time divergent traces.

The main issue of SAT-based model checking, is that traces that either are a Zeno cycle, deadlocking or time divergent, are not part of the (bounded) state space expressed by the SAT-formula. If a property does not hold on such traces, but it does on all others, then the property is considered to be valid for the model. If *all* traces in the model are either Zeno cycles, deadlocking or time divergent, then the set of traces in the (bounded) state space is empty and the property trivially holds for it. This might be perceived as unexpected by the user, and hence it is wise to check whether the model can have such behaviours. Detection by itself is useful, but not sufficient for industrial relevance. A diagnostic means, like a counterexample, is necessary that demonstrates and proves the existence of the trace and also provides useful input towards the engineer to correct the model accordingly.

A more general issue is that of fairness constraints. During our case study (cf. Chapter 6), they could be useful to ensure that components would not starve. They can be expressed at the level of NuSMV, which is not engineer-friendly. An intermediate solution is to adapt the model and ensure a scheduling behaviour where components cannot starve. It however increases the size and behavioural complexity of the model.

4.5 Fault Tree Generation

Fault trees are the artefacts of choice when analysing safety-critical systems [ECSS-Q-ST-40-12C; NASA/SP-2010-580]. Originally developed in 1961 by H.A. Watson to study the Minuteman Launch Control System, it was picked up quickly by Boeing and the nuclear power industry. Fault tree analysis results in a fault tree, which describes how combinations of error events relate to a top-level event, which typically is a fault or failure state of the system. In the original fault tree definition, which we shall refer to as static fault trees [BCS07], the relations are expressed in OR and AND gates. In case of an OR gate, the gate fires if either of its children fire. In case of an AND gate, the gate fires if all its children fire. An extension of static fault trees are dynamic fault trees [BCS07]. Dynamic fault trees provide additional gates and types of basic events, namely cold, warm and hot basic events and priority AND, functionally dependent, voting and spare gates. These additions provide a more fine-grained expression of failing behaviour.

Fault tree analysis is typically performed several times during a space system engineering life-cycle. The first time is during the early phase B, after which the fault tree is subsequently refined. The analysis is often done manually by reasoning top-down over the product tree. Starting from a system-level failure, its possible causes are deduced in terms of subsystem-failures and so on (see also Figure 2.1 for the product tree of a space system). The quality and fidelity of the resulting fault tree depends heavily on the skill of the safety engineer. The effort itself is time-consuming for large and complex systems. Failures may occur at all levels of the system, and may interrelate in unique ways to the system-level failure of interest, and hence require a full system-level understanding of its degraded behaviour. Despite the time-consuming effort of developing a fault tree, its benefits have repeatedly proven itself, and is therefore the safety analysis of preference. The fault tree itself is used for understanding critical events. This itself supports the understanding of design choices and especially its consequences from a safety and dependability perspective. It also enables prioritisation of development, verification and validations efforts, as to put focus on critical parts of the system. It is further amenable to quantification of technical risk through probabilistic risk assessment, and an automated assessment approach is described in Section 4.6.

Finally, during operations it is used as a diagnostic means, helping to understand causes for observed degraded system behaviours during operations.

4.5.1 Approach

Our approach generates a fault tree in an automated manner from a SLIM model, instead of manually analysis and construction. The generation approach is built upon FSAP [BV03a]. The inputs are an extended model (errors events need to be present that affect the system) and a top-level event ϕ. The latter is represented as a propositional property (cf. Section 4.2), and represents a particular (set of) fault/failure state(s) of interest. All paths that lead to a state satisfying ϕ are those leading to the fault/failure state(s) of interest. Error events, from the error model, occurring on those paths are an explanation why ϕ occurred from a fault/failure perspective. A set of such error events is called a *cut set*. The occurrence of the top-level event can be considered as a Boolean formula f where the events in a cut set are conjuncted and where the cut sets themselves are disjunctive. From the set of all paths leading to ϕ, it is possible that not all events in a cut set are causally related to ϕ. It could be that another path exists to a state satisfying ϕ from which a proper subset of the cut set can be derived. Such a smallest subset is called a *minimum cut set*. To determine the minimum cut sets of a fault tree, the Boolean function f is minimised using circuit minimisation algorithms [Weg87]. The resulting function is then represented as a fault tree.

The algorithm for fault tree generation by [BCT07] does not store all paths, as this would become easily intractable memory-wise for larger models. Instead, it adds an additional variable, called a history variable, for each possible error event. The history variable is initially false. When the error event occurs, the associated history variable turns and stays true. Then, for a state that meets ϕ, the cut set can be obtained by looking up the error events for which its history variable is true. Our approach includes the algorithm by [BCT07] which considers ordering of fault events, enabling the generation of priority AND gates (PAND). The technique for detecting the ordering of events is described in [BV03b].

4.5.2 Complexity Analysis

The computation of the cut sets is a specific instance of reachability analysis of $\Diamond\phi$ which has a time complexity of $O(|\mathrm{TS}|)$, where TS is the size of the state space. The state vector for fault tree generation is slightly larger than that for reachability analysis on a SLIM model, as for each error event, an additional history variable is added. Once the cut sets are computed and the Boolean formula f is constructed, this formula is minimised. This is the same as the circuit minimization problem [BCT07] whose time-complexity is known to be NP-hard in the length of the unminimised Boolean formula representing the fault tree [Weg87]. This is typically not a dominating factor, since the size of the state space is typically

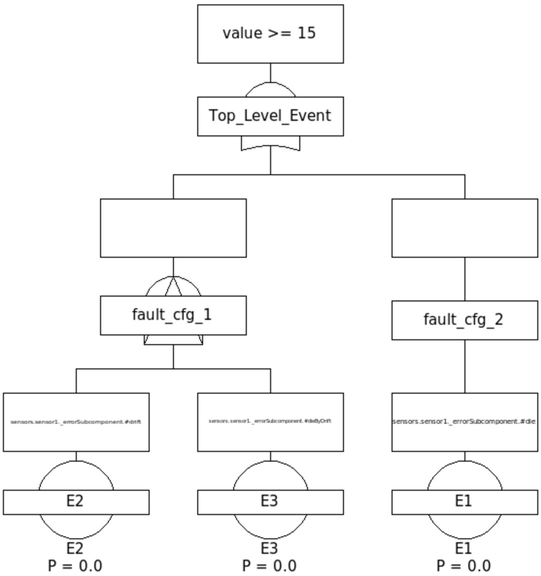

value >= 15
Top_Level_Event
fault_cfg_1
fault_cfg_2
sensors.sensor1._errorSubcomponent.#drift
sensors.sensor1._errorSubcomponent.#dieByDrift
sensors.sensor1._errorSubcomponent.#die
E2
E3
E1
E2
P = 0.0
E3
P = 0.0
E1
P = 0.0

Figure 4.4: Dynamic fault tree generated from the COMPASS toolset. The gate marked with Top_Level_Event is an OR-gate. The gate marked `fault_cfg_1` is a PAND gate. Events E1, E2 and E3 are basic events and correspond respectively to `sensors.sensor1.die`, `sensors.sensor1.drift` and `sensors.sensor1.dieByDrift`.

manifold larger than the size of the unminimised Boolean formula resulting from reachability analysis.

4.5.3 Example

An example fault tree from the sensor-filter model (cf. Appendix A) is shown in Figure 4.4. It is a dynamic fault tree obtained from the top-level property `value >= 15`, a propositional property. It consists of two minimum cut sets, which are called fault configurations. They are $\{E2, E3\}$ and $\{E1\}$. The gate on `fault_cfg_1` is an example of a fault PAND gate. It means that there exists a path towards a state on which `value >= 15` holds, and on that path, event $E2$ occurs before the occurrence of event $E3$. If a static fault tree were generated for the same top-level event, the PAND gate would be an AND gate.

4.5.4 Discussion

A fault tree can be interpreted as an over-approximation of traces that all lead to the top-level event and represented in a compact manner using AND and OR gates. It can be generated using BDD techniques, provided the SLIM model is finite (cf.

Section 4.4), or using SAT-techniques. For the latter, a bound must be given, which may lead to incompleteness of the generated fault tree.

Fault tree analysis is closely related to failure modes and effect analysis, and this relation is discussed in Section 4.8. Fault tree analysis is also closely related to performability analysis, and this relation is discussed in Section 4.12.

In practice, fault trees are developed (manually) in a top-down manner. The occurrence of a top-level event at system-level is described as a fault tree with basic events representing subsystem-level faults. Those basic events are again considered to be top-level events for the subsystem, and in this way, the fault tree can be recursively constructed up to the atomic components. This compositional nature is one reason why fault tree analysis has been successful in industrial practice, since it scales easily with increasing system complexity.

Concerning the generation of fault trees from AADL models, a closely related work is that by [JVB07] and an extension [Li+11] to that also generates PAND-gates. Both uses a different semantics for error modelling and nominal behaviour. In fact, the error behaviour semantics are only interrelated with the system's topology. It does not account for the intertwining with nominal behaviour, and those effects on the fault tree structure. In our approach, we account for the intertwining of error behaviour, nominal behaviour and the system's topology by using the extended model.

4.6 Probabilistic Fault Tree Evaluation

Probabilistic fault tree evaluation computes the probability that the top-level event occurs. This quantification of the technical risk supplements the benefits of fault tree analysis (cf. Section 4.5). In engineering practice, this is also referred to as probabilistic risk assessment.

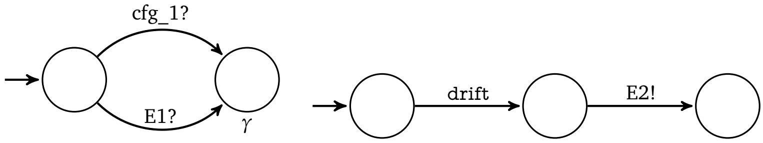
In engineering practice, the probability of the top-level event should be used to compare alternative designs, or ordering criticalities of system elements within the same system. The quantification is therefore a relative measure for *comparing* against a particular baseline. This interpretation comes from the (lack of) accuracy of the used failure rates that are input to fault tree evaluation. A failure rate is the frequency with which a component fails. It is typically expressed in FITs. A FIT is a unit describing the amount of expected failures in one billion hours of operation. While typically the failure rates give rise to a Poisson process describing purely the erroneous behaviour, distributions other than the exponential, e.g. Weibull or Gaussian distributions, are occasionally applied too. Accurate failure rates are difficult to obtain in the space domain. The components tend to be tailored and specialised to the space domain and hence the equipment is manufactured in small batches. Also, they are designed and manufactured with different environmental hazards than consumer electronics. This negates the ability to collect data sets of failing behaviour, and derive statistically confident failure rates. In practice therefore, an estimation of failure rate is used based on anticipated environmental factors, usage characteristics, quality of the manufacturing process and the quality of the used material (cf. [MIL-HDBK-217F]). A conservative attitude is taken in this, leading to rather conservative failure rates. The probabilities of top-level events computed from such sources generally provide an overly pessimistic view, ensuring unaccounted risk is not taken.

4.6.1 Approach

In our approach, all failure rates have to be exponentially distributed, such that a fault tree can be mapped to a continuous-time Markov chain. Inputs are a fault tree, the mission time and the failure rates of the fault tree's basic events. The latter are automatically derived from the SLIM model using the `occurrence` keyword. Then the probability is computed that the top-level event occurs *between* 0 and the given mission time. The latter is typically the anticipated economic lifespan of the system. Using the mapping of dynamic fault trees to Input/Output Interactive Markov Chains (I/O IMC's) [BCS07], the underlying Interactive Markov Chain is obtained. The mapping considers each gate and basic event as a single I/O IMC. It is structured to have input actions which are fire events by children gates/events, and an output action, indicating a fire event by the gate/event. The event of interest is then the output action representing the top-level gate. The state reached by that output action is marked by an unique atomic proposition, let us say γ. A dynamic fault tree is then a set of I/O IMC's that communicate with each other and thus is composed to a single I/O IMC, which again gives rise to an IMC. Weak bisimulation simulation is then performed on the IMC [Her02] to obtain the underlying Continuous Time Markov Chain (CTMC), after which the probability of eventually reaching γ is computed using probabilistic model checking techniques [Bai+03].

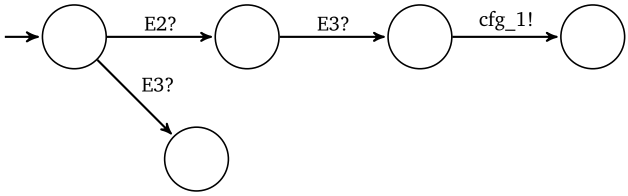
4.6.2 Complexity Analysis

Each basic event and gate in the fault tree is one-to-one mapped to an I/O IMC, resulting in a set of I/O IMCs. As these I/O IMCs are composed together as asynchronous processes that synchronise on their common interaction alphabet, the composed I/O IMC M could be a blow-up process, depending on the fault tree. Weak bisimulation minimisation over M occurs in two phases, namely the computation of the transitive closures of internal transitions followed by partition refinement. The former can be computed in $O(n^{2.376})$ and the latter can be computed in $O(m \cdot \log n)$, where m is number of transitions and n is the number of states [Her02]. The resulting CTMC M' is then subjected to a probabilistic reachability analysis which can be reduced to a transient analysis. We use the Krylov-based method described in Chapter 8 whose time-complexity is discussed in Section 8.3.2.



(a) I/O IMC for the top-level event.

(b) I/O IMC for the basic event of E2.



(c) I/O IMC for the PAND gate.

Figure 4.5: Example I/O IMC's for the fault tree depicted in Figure 4.4.

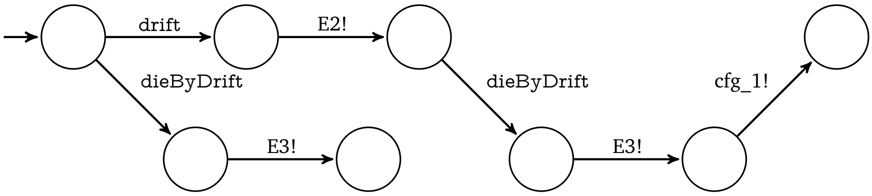


Figure 4.6: Composition of the I/O IMC's associated with basic events E2, E3 and the PAND gate.

4.6.3 Example

Consider the dynamic fault tree of Figure 4.4. The basic event $E2$ is mapped to the I/O IMC shown in Figure 4.5(b). The other basic events are mapped similarly. Fault configuration 1 is mapped to the I/O IMC shown in Figure 4.5(c). The I/O IMC for the top-level event is shown in Figure 4.5(a). When the I/O IMC for fault configuration 1 is composed with the I/O IMC's associated with basic events $E2$ and $E3$, the result is the I/O IMC shown in Figure 4.6. Once all I/O IMC's derived from the fault tree are composed, all action transitions are made hidden and weak bisimulation minimisation is performed. The result is the continuous-time Markov chain shown in Figure 4.7. Given a mission duration parameter t, the probability for the CSL formula $\mathscr{P}_{=p}(\Diamond^{[0,t]}\gamma)$ is computed, which is the probability for the top-level event to trigger.

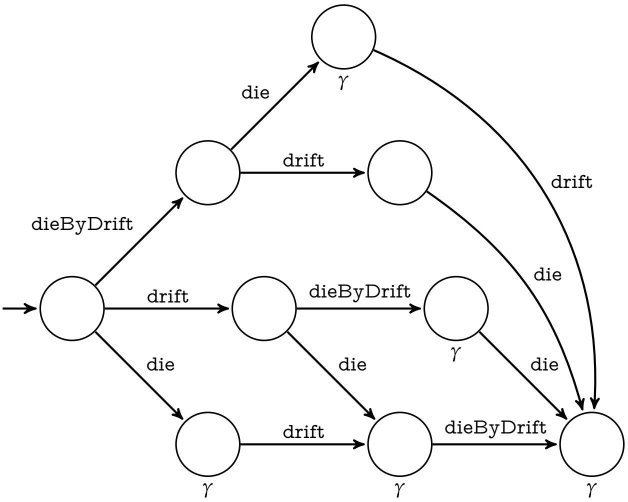


Figure 4.7: Continuous-time Markov chain underlying the fault tree in Figure 4.4. The rates associated with `dieByDrift`, `die`, `drift` are found in Appendix A.

4.6.4 Discussion

Static fault trees are combinatorial, and technically it is not necessary to map them to a Markov chain for their probabilistic evaluation. This allows them for probabilistic distributions other than the exponential one, like a Weibull or Gaussian distribution. The probabilistic evaluation is then simply computing the probabilities of the basic events given a mission time, and multiply/add those discrete probabilities depending whether they are respectively connected to AND or OR-gates. We intentionally restricted ourselves to exponential distributions, allowing us to support dynamic fault trees. In the latter case, dynamic fault trees bring ordering information in addition to the combinatorial information captured by static fault trees. In order to reason probabilistically over ordering of events, the memory-less property of the exponential distribution is convenient.

A major issue with dynamic fault trees is that of non-determinism. Depending on how the dynamic fault tree is structured, weak bisimulation minimisation might not result into a deterministic IMC. This essentially means that depending on the choice of occurrence of events, the probability of the top-level event might differ. Possible scenarios for this are outlined in [Car11]. In our approach, we stop the evaluation and notify the user that the analysis cannot be continued. In 2008, when we designed and implemented dynamic fault tree evaluation, that was state of the art research. In 2010 however, a method was published to compute minimum and maximum probabilities [ZN10]. A minimum probability would indicate that from all the choices that one could take at non-deterministic states, that probability would be the smallest possible probability for the top-levent event to trigger.

The same holds for maximum probabilities and largest possible probability. This work was further extended by [Guc+12] which describes how to compute minimum/maximum expected times and long-run averages on IMCs. In [Car11], the use of these methods was successfully investigated to non-deterministic dynamic fault trees. It is possible to retrofit our approach with these improved methods.

An issue with fault trees and their evaluation is whether the evaluation can be handled compositionally. With static fault trees, this is industrial practice. A system-level fault typically follows the product tree decomposition and when probabilistic risk assessments are conducted, they are typically done at each level of the system. As long as the probabilities were computed with the same mission time, the combinatorial nature of AND and OR-gates allows the probabilities to be combined through respectively multiplication and addition. With dynamic fault trees, this is not possible. As the ordering of events also imply a possibility of different timings between them, the probability of the top-level account need to account for all orderings of events and their possible timings. This information is only present when the full dynamic fault tree exists, hence the probabilistic assessment of dynamic fault trees cannot be trivially divided and composed.

4.7 Probabilistic Fault Tree Verification

Probabilistic fault tree verification is a generalisation of evaluation (cf. Section 4.6). Instead of computing the probability of the top-level to occur, which is a probabilistic existence property (cf. Section 4.2), any arbitrary probabilistic property can be verified. The atomic propositions are expressed over gates. The approach works over both static and dynamic fault trees. It is especially useful to assess whether particular gates fire before other gates, or whether they fire in a particular order. Such assessments have no equivalent in industrial practice.

The approach is similar to probabilistic fault tree evaluation up to the point where atomic propositions are added. For evaluation, there is only one atomic proposition, namely that the top-level event fires. In Section 4.6, this was referred to as γ. For verification, the atomic propositions refer to fault configurations. For example, the probabilistic response pattern (cf. Section 4.2) can be used on the fault tree in Figure 4.4 where $\phi \equiv$ `fault_cfg_1` and $\psi \equiv$ `fault_cfg_2`. These atomic propositions are then added after the respective gate has fired. For ϕ, the state after the cfg_1! event in Figure 4.5(c) would be labelled with the atomic proposition `fault_cfg_1`. The resulting continuous-time Markov chain obtained through weak bisimulation minimisation can differ from the one used for fault tree evaluation due to the difference of labelling.

4.8 Failures, Modes and Effects Table Generation

Failure modes and effects (FMEA) table outlines the possible failure modes and their effects on the system. It is similar to FMECA, but without the criticality analysis for each failure mode and effect combination. It is the inductive counterpart of fault tree analysis. The analysis, which in industrial practice is manually done, starts with identifying the possible failure modes. From these failure modes, the possible effects are analysed. FMECA/FMEA tables, along with fault trees, provide necessary input to the development of the fault management system. Its benefits are similar to those of fault tree analysis, but is different as it is an inductive technique.

4.8.1 Approach

The necessary inputs for FMEA table generation is the extended model, a set of propositional properties that express the possible failure effects, and a cardinality that represents the maximum number of error events for constituting a particular failure mode. Using the same algorithm as for fault tree generation (cf. Section 4.5), the cut sets are determined. For FMEA, this is a bit simpler, since a cardinality is provided which bounds the size of the cut sets. The result is a table in which each row indicates a fault configuration its associated failure effect.

Three options are provided to refine the default generated FMEA table. One is the cardinality. When this is set to 1, the generation considers single-faults and effects only. It is however possible that due to fault management strategies, failure effects only become emergent after multiple failures. By increasing the cardinality, these failure effects can be detected. The second option is the dynamic option. Its purpose is similar to dynamic fault trees, namely considering the ordering of events occurring in a fault configuration. The occurrence order of the resulting failure mode needs to be read from left to right. The underlying algorithm to detect orderings is also the same as for dynamic fault trees. The last option is compactification. It reduces the FMEA table by detecting entries whose fault configuration is a proper subset of another fault configuration with the same failure effect. This implies that other failures are superfluous to the failure effect. Compactifaction weeds these entries out.

4.8.2 Complexity Analysis

FMEA table generation uses the same approach for cut set computation as for fault trees (cf. Section 4.5). The time complexity for this is $O(|\mathrm{TS}|)$, where $|\mathrm{TS}|$ is the size of the state space. All cut sets whose size is larger than the given cardinality are omitted afterwards.

4.8.3 Example

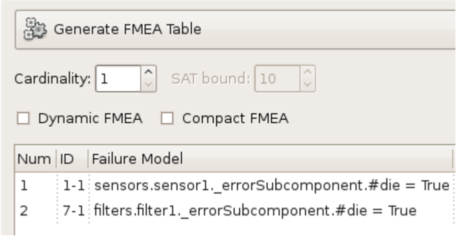
Examples of generated FMEA tables is shown in Figure 4.8. They are all generated from the failure effect `value >= 15 or value = 0`. For cardinality one (cf. Figure 4.8(a)), two entries are shown, namely either first sensor dies, or that the first filter dies. When cardinality two is considered (cf. Figure 4.8(b)), entry 6 shows that the failure effect can also occur by a death by drift and drifting. If the dynamic options would be chosen, the occurrence of this fault configuration would be detected and displayed as `drift & dieByDrift`. The compactified version of Figure 4.8(b) is shown in Figure 4.8(c), and is substantially smaller. Entries like 7 and 8 in Figure 4.8(b) are also covered by the single fault configuration `filters.filter1.die`.

4.8.4 Discussion

FMEA and FTA are closely related. In a typical space system engineering process, they supplement each other as FMEA provides a bottom-up reasoning of failures, whereas FTA provides a top-down reasoning of failures. In the former, the direct failure impact is typically more emphasised, whereas in the latter, the indirect failure impact by fault propagation is accounted for. In this perspective, an FMEA table provides information that is close to understanding local fault effects, and hence allows engineers to design means to detect and mitigate them. A fault tree on the other hand describes how global failure effects are derivable from local faults. The local fault effects are then merely described in higher-level fault effects. In our approach, the top-level events and the local effects are treated equally, and thus only error events that relate to the given effects given by propositional propositions are detected. In [Ern12], a method of impact of isolation over SLIM models is described, which can extract local effects from SLIM models, and its results are more similar to the FMEA tables that are obtained through manual analysis.

4.9 Fault Tolerance Evaluation

Fault tolerance evaluation analyses a set of generated fault trees on their cut sets. For the same system different top-level events could be triggered by the same minimum cut set. To evaluate the fault tolerance, it is desired to understand the *unique* minimum cut sets, and especially their size. In our approach, fault tolerance is computed as the amount of unique minimum cut sets given a *set of fault trees*, and then ordered by their cardinality. For example, for the singleton set consisting of the fault tree depicted in Figure 4.4, there is one unique minimum cut set of cardinality one, and one unique minimum cut set of cardinality two. It can be concluded with the given singleton set, there is no single-fault tolerance. This evaluation is performed in linear time to the number of cut sets in all fault trees.

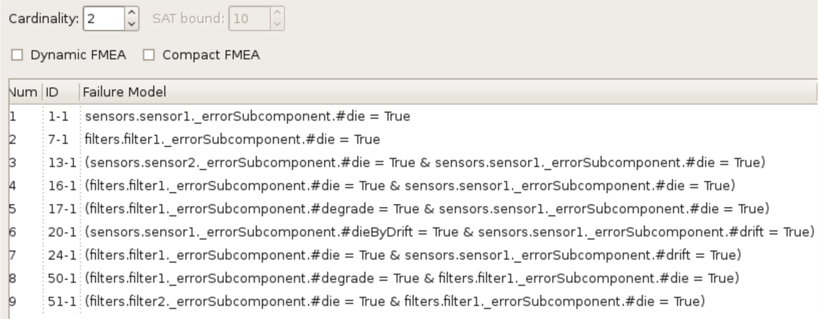


Cardinality: 1 SAT bound: 10

☐ Dynamic FMEA ☐ Compact FMEA

Num	ID	Failure Model
1	1-1	sensors.sensor1._errorSubcomponent.#die = True
2	7-1	filters.filter1._errorSubcomponent.#die = True

(a) Cardinality 1

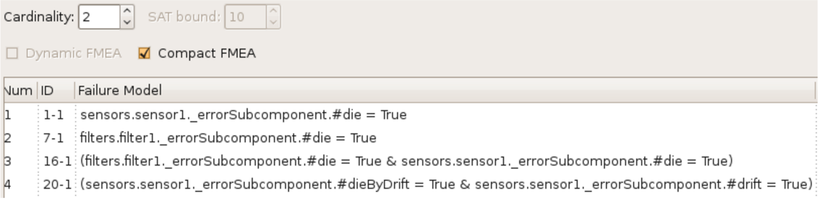


Cardinality: 2 SAT bound: 10

☐ Dynamic FMEA ☐ Compact FMEA

Num	ID	Failure Model
1	1-1	sensors.sensor1._errorSubcomponent.#die = True
2	7-1	filters.filter1._errorSubcomponent.#die = True
3	13-1	(sensors.sensor2._errorSubcomponent.#die = True & sensors.sensor1._errorSubcomponent.#die = True)
4	16-1	(filters.filter1._errorSubcomponent.#die = True & sensors.sensor1._errorSubcomponent.#die = True)
5	17-1	(filters.filter1._errorSubcomponent.#degrade = True & sensors.sensor1._errorSubcomponent.#die = True)
6	20-1	(sensors.sensor1._errorSubcomponent.#dieByDrift = True & sensors.sensor1._errorSubcomponent.#drift = True)
7	24-1	(filters.filter1._errorSubcomponent.#die = True & sensors.sensor1._errorSubcomponent.#drift = True)
8	50-1	(filters.filter1._errorSubcomponent.#degrade = True & filters.filter1._errorSubcomponent.#die = True)
9	51-1	(filters.filter2._errorSubcomponent.#die = True & filters.filter1._errorSubcomponent.#die = True)

(b) Cardinality 2



Cardinality: 2 SAT bound: 10

☐ Dynamic FMEA ☑ Compact FMEA

Num	ID	Failure Model
1	1-1	sensors.sensor1._errorSubcomponent.#die = True
2	7-1	filters.filter1._errorSubcomponent.#die = True
3	16-1	(filters.filter1._errorSubcomponent.#die = True & sensors.sensor1._errorSubcomponent.#die = True)
4	20-1	(sensors.sensor1._errorSubcomponent.#dieByDrift = True & sensors.sensor1._errorSubcomponent.#drift = True)

(c) Cardinality 2 and compactification

Figure 4.8: Generated FMEA tables from the failure effect `value >= 15` **or** `value = 0`. In the screenshots, the failure effect column is omitted for readability.

4.10 Diagnosability

When a system is monitored, by either an operator or another system, it is important to know whether sufficient information is exposed by the system for the monitor to derive valid conclusions regarding a system's state. For spacecraft operators, the available means for (correct) telemetry are essential for obtaining a quick understanding of the system under operation. If particular telemetry is not available, a memory dump of the spacecraft needs to be commenced, and the dump, which is typically sizeable, needs to be analysed. The additional time and resources needed puts pressure on the time-constrained schedule of spacecraft operators. In case the system (i.e. the plant) is monitored by another system, e.g. a fault management system, then having sufficient observables is essential for distinguishing failures and nominal from non-nominal states. If insufficient observables are designed for during the design of the nominal behaviour, then observability issues typically arise during the design of the fault management system. In the worst-case, the nominal design has to be modified, whose changes ripple through (and possible delay) the overall system development schedule. For this reason, the nominal system is typically designed to be overly diagnosable, i.e., there are more observables present than needed to distinguish failures, nominal and non-nominal behaviour. This however adds unnecessary design complexity.

Diagnosability analysis can be used to check whether a model representing the system provides sufficient information for distinguishing a diagnosis condition of interest. If it is, alternative less complex models with different (and perhaps less) observables can be constructed and analysed, to see whether they provide sufficient diagnosability characteristics. In case the system is insufficiently diagnosable, the analysis provides an counterexample. It can be used to understand which parts of the design are responsible for the insufficient diagnosability.

4.10.1 Approach

We perform diagnosability analysis by the approach of [CPC03]. Data ports tagged with `observable` are taken as the points which can be used by the observer (e.g. operator or a monitoring system) to infer a diagnosis. Let $O = \{o_1, \ldots, o_n\}$ be a set of all observable data ports occurring in the model. A particular condition that has to be diagnosed is provided as a propositional property ϕ, e.g. the occurrence of a failure. The diagnosis condition is to distinguish the occurrence of a failure ϕ from the absence of a failure $\neg\phi$ using only information derived from the observables. This can be computed by partitioning the set of all reachable states into two sets, namely the states that satisfy ϕ and the states that satisfy $\neg\phi$. If the former set does not intersect with the latter set with respect to the valuations on O, then we can conclude that ϕ is diagnosable. If however there is a non-empty intersection, then it means there are two reachable states which have the same valuation on O, but that one state satisfies ϕ and the other satisfies $\neg\phi$. Such a counterexample can be provided by two traces from the initial state to the two respective states.

The algorithmic approach towards this is by coupled reachability. Given a model M which has O as the set of observables and ϕ as the condition to be diagnosed. A twin model M' is constructed from M which is identical in structure as M, but whose state variables are different (e.g. primed) from M. This means M' has a set of (primed) observables O'. To prove whether ϕ is diagnosable, one verifies a reachability property on the asynchronous composition of M and M', namely $M||M' \models \Diamond(o_1 = o'_1 \wedge \cdots \wedge o_n = o'_n \wedge \phi \wedge \neg\phi')$. If the property holds on $M||M'$, then it means ϕ is not diagnosable in M. The trace towards the reached state is the counterexample. The unprimed variables in the trace represent the part to which ϕ holds, whereas the primed variables represent the part to which $\neg\phi'$, while having the same valuation to the observables.

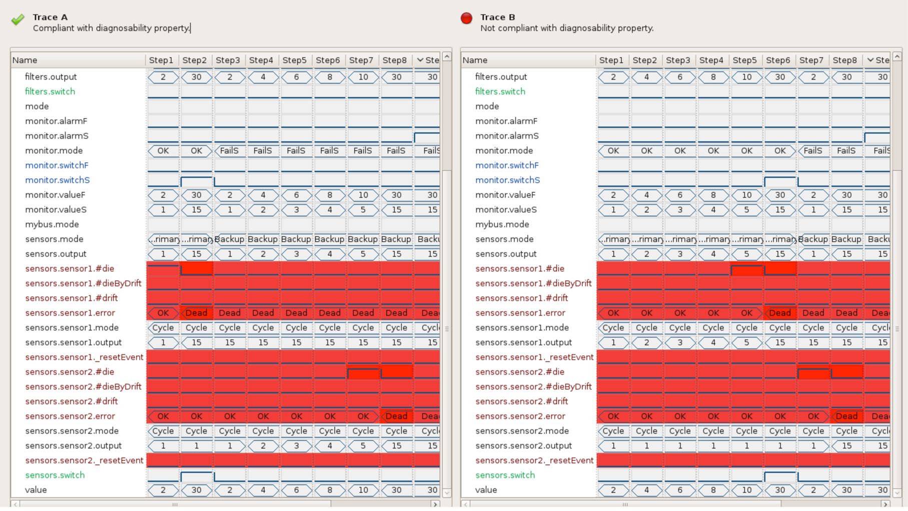
The space and time-complexity for diagnosability analysis is relatively high due to the asynchronous composition of M and M' The user can provide two additional inputs to filter and optimise the analysis. First, the diagnosis context can be provided. It is an additional propositional property ψ and has to hold on all states for which the diagnosis condition is checked, i.e. $M||M' \models \Diamond(\psi \wedge \psi' \implies (o_1 = o'_1 \wedge \cdots \wedge o_n = o'_n \wedge \phi \wedge \neg\phi'))$. This filters possible false undiagnosability reports, as it could be the case that a diagnosis condition needs only to hold in particular system modes (e.g. safe modes). Second, path restrictions can be provided. These are additional propositional properties, that are provided in an order of interest. Diagnosability analysis will only traverse the part of the state space for which these propositional properties hold in their occurring order.

4.10.2 Complexity Analysis

Diagnosability analysis is reduced to coupled reachability. Reachability by itself is $O(|TS|)$, where $|TS|$ is the size of the state space. In the coupled case, it means that state vector is doubled in size. Also, the amount of parallel processes doubles. In the worst-case, i.e., when no observables are present, no interaction occurs between M and M', leading to a state space of size $2 \cdot |TS|$. Hence the worst-case time-complexity of coupled reachability is $O(|TS|)$. In practice, diagnosability analysis is performed on observables that are synchronised. The resulting state space is smaller than $2 \cdot |TS|$.

4.10.3 Example

For the sensor-filter case study in Appendix A, there are two observables, namely `alarmF` and `alarmS`. From the fault injections in Section 4.1, let us consider only the ones on the first and second sensors. If we would like to check whether a sensor failure is diagnosable, the propositional property `value >= 15` is provided as a diagnosis condition. The reachability property for diagnosability becomes $\Diamond(\texttt{alarmF} = \texttt{alarmF}' \ \wedge\ \texttt{alarmS} = \texttt{alarmS}' \wedge \texttt{value >= 15} \ \wedge\ \neg(\texttt{value}'\texttt{>= 15}))$. The property holds (cf. Figure 4.9), as there is a state in M for which the first sensor has died, but where in M' this has not happened. Yet, their observables



Trace A
Compliant with diagnosability property.

Name	Step1	Step2	Step3	Step4	Step5	Step6	Step7	Step8	Ste
filters.output	2	30	2	4	6	8	10	30	30
filters.switch									
mode									
monitor.alarmF									
monitor.alarmS									
monitor.mode	OK	OK	FailS	FailS	FailS	FailS	FailS	FailS	Fail
monitor.switchF									
monitor.switchS									
monitor.valueF	2	30	2	4	6	8	10	30	30
monitor.valueS	1	15	1	2	3	4	5	15	15
mybus.mode									
sensors.mode	...rimary	...rimary	Backup	Backup	Backup	Backup	Backup	Backup	Backu
sensors.output	1	15	1	2	3	4	5	15	15
sensors.sensor1.#die									
sensors.sensor1.#dieByDrift									
sensors.sensor1.#drift									
sensors.sensor1.error	OK	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dea
sensors.sensor1.mode	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycl
sensors.sensor1.output	1	15	15	15	15	15	15	15	15
sensors.sensor1._resetEvent									
sensors.sensor2.#die									
sensors.sensor2.#dieByDrift									
sensors.sensor2.#drift									
sensors.sensor2.error	OK	OK	OK	OK	OK	OK	OK	Dead	Dea
sensors.sensor2.mode	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycl
sensors.sensor2.output	1	1	1	2	3	4	5	15	15
sensors.sensor2._resetEvent									
sensors.switch									
value	2	30	2	4	6	8	10	30	30

Trace B
Not compliant with diagnosability property.

Name	Step1	Step2	Step3	Step4	Step5	Step6	Step7	Step8	Ste
filters.output	2	4	6	8	10	30	2	30	30
filters.switch									
mode									
monitor.alarmF									
monitor.alarmS									
monitor.mode	OK	OK	OK	OK	OK	OK	FailS	FailS	Fail
monitor.switchF									
monitor.switchS									
monitor.valueF	2	4	6	8	10	30	2	30	30
monitor.valueS	1	2	3	4	5	15	1	15	15
mybus.mode									
sensors.mode	...rimary	...rimary	...rimary	...rimary	...rimary	...rimary	Backup	Backup	Backu
sensors.output	1	2	3	4	5	15	1	15	15
sensors.sensor1.#die									
sensors.sensor1.#dieByDrift									
sensors.sensor1.#drift									
sensors.sensor1.error	OK	OK	OK	OK	OK	Dead	Dead	Dead	Dea
sensors.sensor1.mode	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycl
sensors.sensor1.output	1	2	3	4	5	15	15	15	15
sensors.sensor1._resetEvent									
sensors.sensor2.#die									
sensors.sensor2.#dieByDrift									
sensors.sensor2.#drift									
sensors.sensor2.error	OK	OK	OK	OK	OK	OK	OK	Dead	Dea
sensors.sensor2.mode	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycle	Cycl
sensors.sensor2.output	1	1	1	1	1	1	1	15	15
sensors.sensor2._resetEvent									
sensors.switch									
value	2	4	6	8	10	30	2	30	30

Figure 4.9: Proof that `value >= 15` cannot be diagnosed on the sensor-filter model. At the sixth state (step 2), the value of the left side satisfies the diagnosis condition, whereas the right side does not. Yet, they have the same valuations for the observables (not visible in the screenshot).

are the same. The observable `alarmS` is however only raised when the second sensor has failed, meaning the diagnosis context should be that the second sensor is active, i.e. ψ is sensors.**mode** = **mode**:Backup. When this diagnosis context is used, we find that the model is diagnosable.

4.10.4 Discussion

For hybrid models, one must be careful when working with assignments to observable variables (Boolean data ports or data subcomponents) that refer to variables with a continuous domain. For instance, if a certain continuous value x goes from -1 to 1, it crosses the value 0. If the diagnosability condition depends on the fact that $x = 0$, and x is tested in an assignment to an observable data port p, the observation that $x = 0$ might go by unnoticed. The reason for this is that the discretization of the continuous variable for delivery to the data port happens non-deterministically at random intervals, which means that it is possible but not guaranteed that p detects the value $x = 0$. If this however is explicitly desired, discretization at $x = 0$ must be manually specified in the SLIM model. This can be accomplished, for example, by splitting the mode that defines the dynamics of x into two sub-modes where the first is taken for non-positive values of x (by requiring the invariant $x <= 0$), and is left when the value $x = 0$ is reached. To this aim, the corresponding transition to the second sub-mode is guarded by the condition $x = 0$, and sets p to the value true.

Diagnosability analysis is, compared to the other analyses, expensive due its use of coupled reachability. If the state space of the system is large, then typically that of coupled reachability is significantly larger. This is a concern for scalability. In recent literature the issue of scalability has been tackled [SP07]. That solution however only applies if the system satisfies particular preconditions on their inter-component communication, like that all variables are accessed within their strictly hierarchical scope. SLIM models, with their data port connections, do not satisfy this condition by definition.

Our approach only handles direct diagnosability. For direct diagnosability, there is a bijective relation between the diagnosis condition and the values of observables. This bijection holds for each state in the state space (or its subspace in case path/context constraints are provided). For fault management, if the occurrence of a failure does not *directly* trigger a change of observables, the failure cannot be diagnosed using direct diagnosability. It can be using delayed diagnosability, or sometimes called Δ-diagnosability conditions [Tri02]. In such an approach, a Δ is provided, which is a time window in which a diagnosis condition should become observable. In our approach, Δ-diagnosability conditions cannot be handled.

Our approach to diagnosability analysis is formulated as a decidability problem: whether a system is, or not, diagnosable. During the project, the idea was coined, whether the issue of diagnosability could be presented as a synthesis problem: assuming a system that is overly diagnosable, which minimum set of observables are needed to infer correctly a set of diagnosis conditions? This synthesis problem has been explored in [Bit+11a].

4.11 Fault Management Effectiveness

Fault management effectiveness analyses provide insight how and whether a system, in the presence of errors and a fault management system, handles these events effectively. The provided analyses are organised according to the FDIR-paradigm, and offer a method to analyse the effectiveness of failure detection, fault isolation and failure recovery. The analyses specialise existing analysis techniques as model checking and fault tree generation by tailoring them as solutions to problems encountered during the development of fault management systems. They have no direct counterpart in the space system engineering life-cycle.

4.11.1 Approach

Fault detection analysis questions, given a propositional property ϕ, which observables $O_\phi \subset O$, with O the set of all observables, change value with respect to their default value in their initial state I. When ϕ expresses the occurrences of a fault or failure, then the resulting observables are its possible detection means. It can be casted as a model checking problem, namely by checking the property

$\Box(\phi \implies (\phi \bigcup (o = \neg I(o))))$, where o is an observable and $\neg I(o)$ is the negation of observable o's initial value. When the property holds for observable o, then it means observable o is (part of) the detection means for the event expressed by ϕ.

Fault isolation analysis questions, given all observables in the model, what (set of) error events trigger each observable. The result of this analysis is a fault tree for each observable, and where the basic events are error events that through AND and OR-gates can trigger the observable. For perfect isolation, each observable is triggered by only one error event. Fault isolation analysis rehashes fault tree generation by generating a fault tree for each $o \in O$ with the top-level event the propositional property $o = \neg I(o)$.

Fault recovery analysis questions whether a recovery property holds. The property is expressed using the property patterns as discussed in Section 4.2. An additional pattern is supported here, which is named *existential response*. It is typically used when a fault or failure occurs, whether there is a path towards a recovered state. This contrasts with global response, where all paths would need to lead to a recovered state. Upon violation of a property, a counterexample is presented. Fault recovery analysis is therefore the same as model checking.

4.11.2 Complexity Analysis

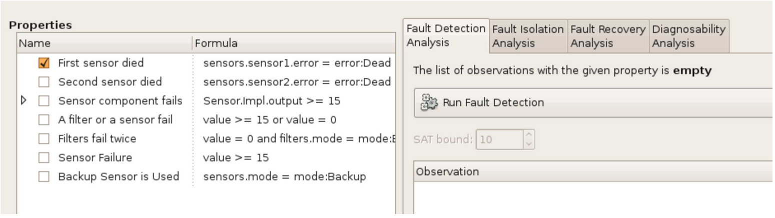
Fault detection analysis and fault recovery analysis are mapped upon model checking. The complexity of that is discussed earlier in this chapter (cf. Section 4.4.2). Fault isolation analysis is mapped upon fault tree generation. The associating complexity is discussed in Section 4.6.2.

4.11.3 Example

Consider the sensor-filter model (cf. Appendix A) with the fault injections from Section 4.1.3 applied. If we would compute the observables for the death of the first sensor, i.e. sensors.sensor1.**error** = **error**:Dead, the list of observables is empty (cf. Figure 4.10(a)). If we would do the same for the death of the second sensor, i.e. sensors.sensor2.**error** = **error**:Dead, the observable alarmS would be triggered (cf. Figure 4.10(b). This is expected behaviour, as the sensor alarm is only raised by the monitor when the redundant sensor has failed too.

Fault isolation on the sensor-filter model results in two fault trees, as there are in total two observables present in the model. For each fault tree associated with an observable, its top-level event is the value change of the observable itself. That means for the sensor alarm and the filter alarm, they are respectively alarmS = **true** and alarmF = **true**. An example of the fault tree for alarmS = **true** is shown in Figure 4.11.

For an example of fault recovery, we refer the reader to Section 4.4.3, which demonstrates model checking on a typical recovery property.



Properties

	Name	Formula
☑	First sensor died	sensors.sensor1.error = error:Dead
☐	Second sensor died	sensors.sensor2.error = error:Dead
▷ ☐	Sensor component fails	Sensor.Impl.output >= 15
☐	A filter or a sensor fail	value >= 15 or value = 0
☐	Filters fail twice	value = 0 and filters.mode = mode:
☐	Sensor Failure	value >= 15
☐	Backup Sensor is Used	sensors.mode = mode:Backup

Fault Detection Analysis
Fault Isolation Analysis
Fault Recovery Analysis
Diagnosability Analysis

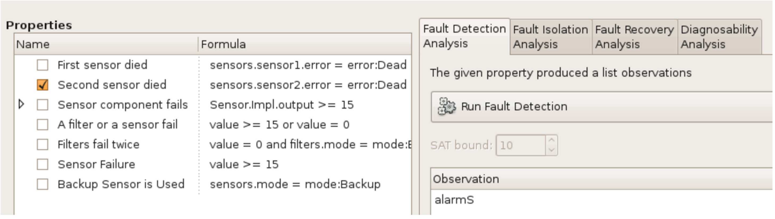
The list of observations with the given property is **empty**

Run Fault Detection

SAT bound: 10

Observation

(a) The set of observables is empty for the occurrence death of the first sensor.



Properties

	Name	Formula
☐	First sensor died	sensors.sensor1.error = error:Dead
☑	Second sensor died	sensors.sensor2.error = error:Dead
▷ ☐	Sensor component fails	Sensor.Impl.output >= 15
☐	A filter or a sensor fail	value >= 15 or value = 0
☐	Filters fail twice	value = 0 and filters.mode = mode:
☐	Sensor Failure	value >= 15
☐	Backup Sensor is Used	sensors.mode = mode:Backup

Fault Detection Analysis
Fault Isolation Analysis
Fault Recovery Analysis
Diagnosability Analysis

The given property produced a list observations

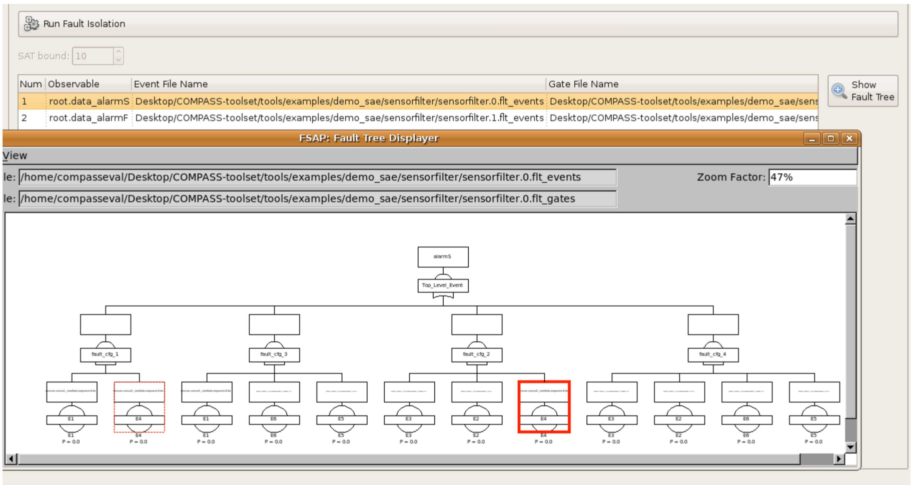
Run Fault Detection

SAT bound: 10

Observation
alarmS

(b) The observable `alarmS` is triggered after the death of the first sensor.

Figure 4.10: Fault detection analysis.



Run Fault Isolation

SAT bound: 10

Num	Observable	Event File Name	Gate File Name
1	root.data_alarmS	Desktop/COMPASS-toolset/tools/examples/demo_sae/sensorfilter/sensorfilter.0.flt_events	Desktop/COMPASS-toolset/tools/examples/demo_sae/sens
2	root.data_alarmF	Desktop/COMPASS-toolset/tools/examples/demo_sae/sensorfilter/sensorfilter.1.flt_events	Desktop/COMPASS-toolset/tools/examples/demo_sae/sens

Show Fault Tree
FSAP: Fault Tree Displayer
View

le: /home/compasseval/Desktop/COMPASS-toolset/tools/examples/demo_sae/sensorfilter/sensorfilter.0.flt_events

le: /home/compasseval/Desktop/COMPASS-toolset/tools/examples/demo_sae/sensorfilter/sensorfilter.0.flt_gates

Zoom Factor: 47%

alarmS
Top_Level_Event
fault_cfg_1
fault_cfg_3
fault_cfg_2
fault_cfg_4

Figure 4.11: Fault isolation of the observable `alarmS`.

4.11.4 Discussion

Fault detection and isolation analysis bear a resemblance to diagnosability analysis (cf. Section 4.10). There are however distinct differences, and depending on the objective, either analysis could be preferred over another.

The output of failure detection analysis provides less information than diagnosability analysis. With fault detection analysis, the set of triggered observables does not imply that that set is bijective with respect to failure occurrences. Depending on the model, fault detection analysis can generate the same set of observables for another failure, which would be a proof of undiagnosability. The proof however does not state the causal reasons for undiagnosability, contrary to diagnosability analysis. Fault detection analysis is however less resource-consuming as it does not require a coupled reachability analysis.

The output of fault isolation analysis also provides less information than diagnosability analysis. If fault tree isolation analysis shows that an observable is only triggered by a single fault, then there is a bijective relation of that fault with an observable, leaving no doubt about the degree of diagnosability. If a system triggers multiple observables upon occurrence of faults, and that a set of observables are needed to infer a correct diagnosis, then the resulting fault trees must be studied whether each combination of observables has an unique fault mapping. This allows one to decide whether a system is diagnosable. However, if it is not diagnosable, fault isolation analysis does not provide a causal explanation of non-diagnosability.

4.12 Performability Analysis

The combination of performance evaluation and dependability analysis is performability [Mey92]. Performance evaluation quantifies how well the system does an activity or a function. Dependability encompasses the concepts availability, reliability and maintainability. Reliability for example concerns itself purely with occurrence of failures, and typically how often they occur. Using failure rates, and use them in a modelling and analysis formalism like reliability block diagrams [MIL-HDBK-338B], Markov models, Monte Carlo simulations or fault trees, it is determined how composite systems, and in particular the overall system can fail, and how often. Availability on the other hand concerns itself purely with system downtime and uptime, and the typical measure of interest is its ratio. It is typically computed by $\mathrm{MTBF}/(\mathrm{MTBF}+\mathrm{MTTR})$, where MTBF is the mean time between failures and the MTTR is the mean time to repair. Other factors that do not relate to failures but impact system functions like long start-up times, or short operational time windows, also may impact availability. Maintainability concerns itself with restoring the system to an operational state. A typical measure that relates to failures is the MTTR. It is not strictly related to failures, as maintainability concerns can also originate from changes of the system's operational environment and its necessary adaptations for it. Dependability tends to overlap with safety if its aspects, e.g. unavailability and failures, impact safety concerns as human life, the environment, property, spacecraft, launcher and ground systems.

When these aspects are interrelated due to failures, we get the notion of performability, which intuitively is captured as the performance under degraded modes of operation. Its analysis is vital to systems which are designed with a degree of fault tolerance, e.g. a redundant processor system with multiple processor modules, or four-axis gyroscopes where the fourth axis can compensate for either of the other three axes. These degradable system can also be higher-level, like a full mission, where there is a satellite constellation, of which the satellites are fault tolerant to a certain degree, but that after a satellite has failed, a spare satellite needs to activated, the spare needs to be replenished in due time, and that there are several strategies for replenishment. What would be the availability in the presence of such tolerances to various faults, failures and possible repair strategies? Typical modelling formalisms in literature suitable for such analysis objectives are stochastic (timed) Petri nets and Markov reward models. In space engineering practice, it is typical to use Monte-Carlo simulations over Simulink models, or in case a probabilistic measure of performability needs to be obtained, stochastic timed Petri nets.

4.12.1 Approach

Our approach towards performability is by computing a probabilistic measure over the Markov model underlying the state space of the extended model. The needed inputs are an extended SLIM model and a probabilistic property expressed in CSL ϕ. The latter is obtained through the probabilistic patterns described in Section 4.2. Initially, the full discrete state space of the extended model is generated, i.e. the transition system $\mathrm{TS} = (Cnf, \kappa_0, L, \Longrightarrow)$ resulting from the network of event-data automata. The transitions $\Longrightarrow$ are then partitioned in to two non-overlapping sets, namely Markovian transitions and interactive transitions. Markovian transitions are those triggered by error events that have an `occurrence` rate associated. All other transitions are interactive transitions. The propositions from ϕ are attached to the states (in *Cnf*) satisfying them. The result is an interactive Markov chain [Her02]. The typical state space of a realistic industrially sized model is large, and hence first stochastic weak bisimulation is performed. All action transitions are made internal before this happens. The resulting is a smaller IMC that preserves the (dis)satisfaction of ϕ. If it does not contain internal non-determinism, then the IMC is equivalent to a continuous-time Markov chain, upon which we model check ϕ. We used a Krylov-based method for this (cf. Chapter 8), as its numerical stability is better for occurring rates with large disparities. Then probabilistic model checking is performed for varying upper time bounds from which we plot a cumulative distribution function.

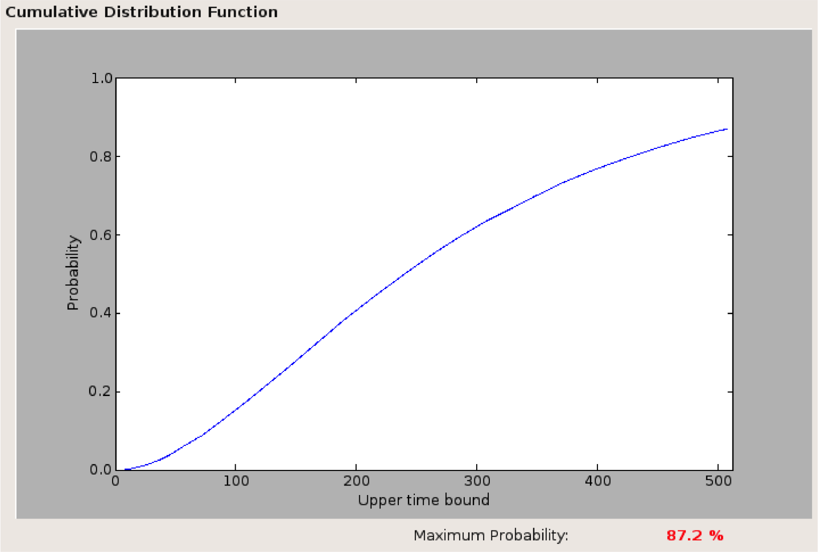


Figure 4.12: Cumulative distribution function between 0 and 512 of $\mathscr{P}_{=1-p}(\neg\phi \bigcup^{[0,512]} (\neg\phi \wedge \psi))$ on the Markov chain depicted in Figure 4.13.

4.12.2 Complexity Analysis

Perfomability analysis starts with the generation of the state space, which is formally characterised as the tuple $TS = (Cnf, \kappa_0, L, \Longrightarrow)$. Weak bisimulation minimisation is then performed over it. As discussed in Section 4.6.2, the time complexity of that is $O(n^{2.376} + m \cdot \log n)$. Note that contrary for fault tree evaluation, $m = |\Longrightarrow|$ and $n = |Cnf|$. This is significantly more complex than for fault tree evaluation where the underlying IMC is derived from the fault tree. The CTMC resulting from weak bisimulation minimisation is then subjected to probabilistic model checking of the CSL property ϕ. The time-complexity of verifying ϕ is dependent of its length in terms of subformula's and the time needed for transient analysis [Bai+03]. Our approach uses the probabilistic patterns (cf. Section 4.2). In this pattern system, either there are no subformula's, or there is at most subformula (e.g. probabilistic response pattern), thus bounding the length of ϕ to 2. The time-complexity for computing the transient using Krylov subspace methods is discussed in detail in Section 8.3.2.

4.12.3 Example

An example cumulative distribution function from the sensor-filter model (cf. Appendix A) is shown in Figure 4.12. It is obtained from the probabilistic precedence pattern stating that the sensor bank fails, i.e. ϕ is sensors.sensor2.**error** $=$

`error`:Dead, before the filter bank dies, i.e. ψ is filters.filter2.`error` = `error`:Dead, within 512 time units (cf. Section 4.2.3). Its underlying interactive Markov chain consists of 161 states. After weak bisimulation minimisation, the Markov chain is reduced to 18 states. The result is shown in Figure 4.13. As the Markov chain does not contain internal transitions, it can be interpreted as a continuous-time Markov chain. On this Markov chain, we check the property $\mathscr{P}_{=1-p}(\neg\phi \bigcup^{[0,512]}(\neg\phi \wedge \psi))$, which is the CSL property underlying the probabilistic precedence pattern, and then plot the cumulative distribution function as shown in Figure 4.12.

4.12.4 Discussion

In our approach, performability analysis and fault tree evaluation are based on the system's behaviour captured by the state space and as such they bear a strong resemblance. Their main difference lies in their degree of abstraction. Whereas fault tree generation abstracts the state space into a tree shape, performability analysis considers a more fine-grained abstraction of the state space using a notion of stochastic weak bisimulation equivalence on the state space of the extended model. Both abstractions have their drawbacks. Fault tree generation generates at most PAND-gates for capturing orderings of events. Performability takes more subtle orderings in account, like those induced by repairs. Even cyclic behaviours can be accounted for in performability. The drawback is the increased time and space complexity. Performability generates the full state space, after which it performs stochastic weak bisimulation minimisation. The resulting IMC is typically smaller, so the time needed for transient analysis (cf. Chapter 8) over it is typically negligible. Note that in industrial practice, the relation between performability analysis and fault tree evaluation is typically not so obvious, given that for both analyses specialised models are manually developed and analysed, and that their (only) commonality is that use the technical documents as inputs.

Non-determinism was also, akin to fault tree evaluation in Section 4.6, an issue with performability. If the IMC after weak bisimulation minimisation contains non-determinism, then we stop the analysis. Akin to fault tree evaluation and verification (cf. Sections 4.6 and 4.7 for the discussion on non-determinism in that context), this was state of the art in 2008, when we developed our approach towards performability. By using the works by [ZN10] and by [Guc+12], it is now possible to compute minimum/maximum probabilities and expected time instead.

Performability cannot be performed on hybrid SLIM models. The problem to this is fundamental in nature. The state spaces by hybrid SLIM models have three types of transitions: action, Markovian or timed. The combination of action and Markovian transitions are interactive Markov chains. The combination of action and timed transitions are timed automata. However, for state spaces that have both timed and Markovian transitions, no underlying formalism exists that have semantics that align with expected system behaviour that is both consistent and complete. More specifically, consider a state s which has a Markovian transition towards state s' and a guarded time transition to state s'' and both transitions

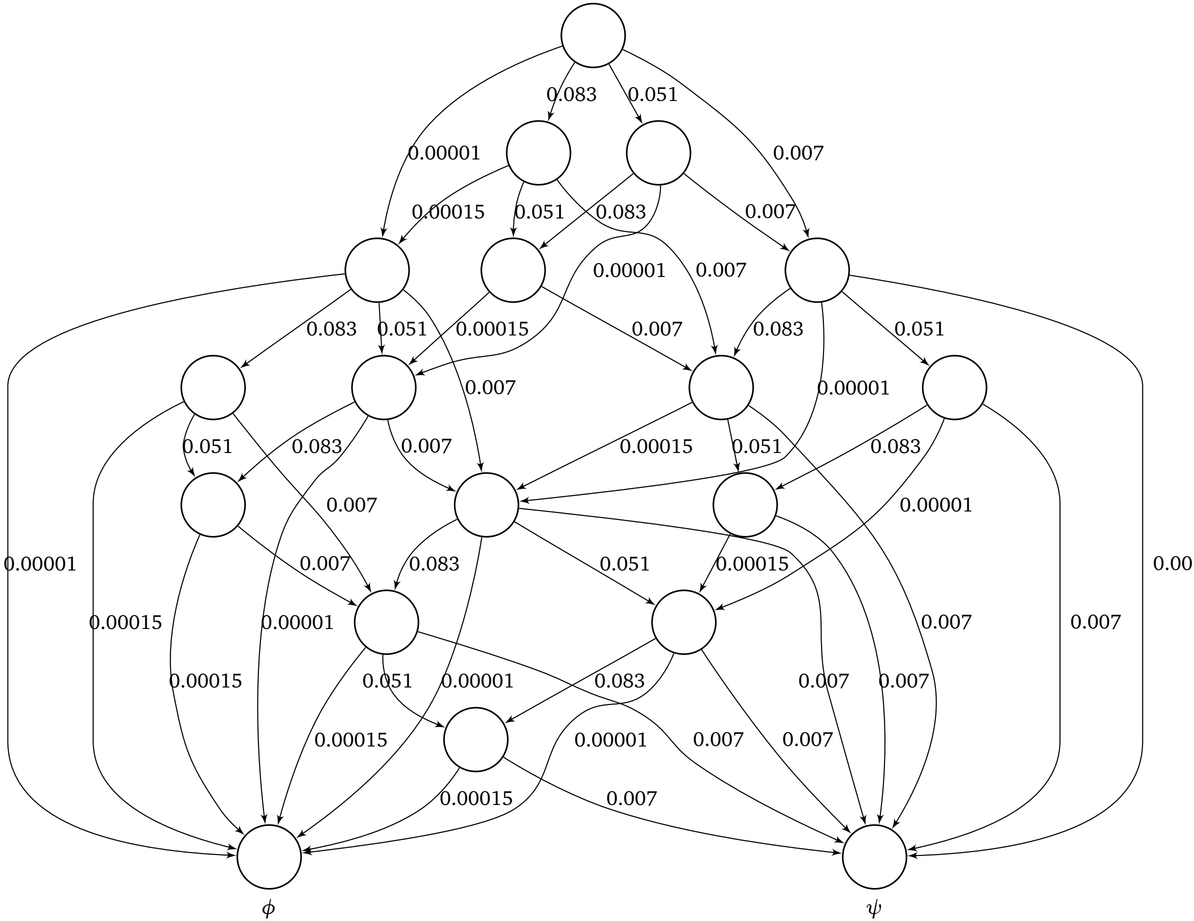

0.083
0.051
0.00001
0.007
0.00015
0.051
0.083
0.007
0.00001
0.007
0.083
0.051
0.00015
0.007
0.083
0.051
0.007
0.00001
0.051
0.083
0.007
0.00015
0.051
0.083
0.007
0.00001
0.00001
0.007
0.083
0.051
0.00015
0.007
0.00015
0.00001
0.007
0.007
0.00015
0.051
0.00001
0.083
0.007
0.007
0.00015
0.00001
0.007
0.007
0.00015
0.007
ϕ
ψ

Figure 4.13: Continuous-time Markov chain obtained for performability analysis. The upper state is the initial state.

are enabled. It is unclear how the transient should be computed on models that capture that.

During our modelling efforts, we observed that our approach towards performability cannot handle reactive systems. In reactive systems, there is an cycle of action-transitions present in the state space. In the IMC formalism, there is the assumption that any action-transition is always taken over a Markovian transition, which is called the maximum progress assumption. When this assumption is applied to a reactive system, the only behaviour that remains is the original nominal behaviour. Degraded behaviour, which can occur from Markovian transitions that were induced by faults, are unreachable due to the maximum progress assumption. We tried to resolve this by replacing all action-transitions with a very fast rate which would represent a near instantaneous probabilistic transition. If the rate is sufficiently large, its impact on the computed probabilities is small. Since the replacement rate should be very fast compared to the other occurring rates, the problem of stiffness kicks in, and in particular how to deal with it in a numerical stable way. Then secondly, the replacement rate affects the resulting probabilities, although the impact is small, it is an open question how to quantify this. We therefore consider performability analysis for reactive systems still an open problem.

5

Inside the COMPASS Toolset

The COMPASS toolset is an integration of various existing software components and additional software to handle previously non-existing functionality. The two main software components, the NuSMV model checker and the Markov Reward Model Checker (MRMC), are its core engines. They contain the algorithms that perform the time and memory consuming analyses. The amount of lines of additional code are near the 100,000, and was written within two years, by 16 different developers located at three development sites in three countries, using five different source languages. The overall architecture of COMPASS was aimed with the primary values of reuse, enable effective remote cooperation between the developers and meeting the project deadlines.

This chapter presents an overview of the COMPASS's internal structure with in particular the core components that do the parsing, checking and transformations to and from the existing software components.

5.1 Building Blocks

The component diagram of the core components in the COMPASS toolset is shown in Figure 5.1. The reading direction is from left to right and from top to bottom.

SLIM compiler	Parses and checks nominal and error models according to the SLIM language specification outlined in [Nol11b]. The compiler is based on the ANTLR parser generator [Par07]. The target language is Python, and the SLIM itself is then transformed to a object-oriented datastructure called `CompileInfo`. The `CompileInfo` represents the full SLIM model.

Model extension	The `CompileInfo` datastructure is also the input and the output for model extension. It is written in Python. Its main function is to apply fault injections, specified in a XML format whose meta-model is captured by a XSD (provided in the COMPASS toolset), to link the parsed nominal and error models and creating the extended model. The latter is represented as an object of `CompileInfo` as well.
SLIM to SMV	For all analyses, the `CompileInfo` data-structure is transformed to a SMV model, which is the input to the NuSMV model checker [Cim+02]. This component, called SLIM to SMV, is written in Python, and handles both extended models and purely nominal models.
Property manager	Parses properties specified using the patterns (cf. Section 4.2) and written in a XML format whose meta-model is captured by a XSD (also provided in the COMPASS toolset). The patterns itself are also described by a XML format whose meta-model is also captured by a XSD. This allows for flexibility on adding new patterns when fit. The property manager can handle the generation of LTL, CTL and CSL formulae in respectively textual NuSMV and MRMC format.
NuSMV	This model checker is one of the core engines that implements the model checking algorithms, and providing a framework for many other analyses supported by the COMPASS toolset. Additions and enhancements were made to meet the project requirements. Existing FSAP-functionality [BV03a] generating safety and dependability artefacts like fault trees and FMEA tables was slightly modified to handle the arbitrary structure of the error model transition systems. Also two additional modules were developed, one for handling the FDIR analyses, and one providing a cornerstone block in performability analysis, named *SMV to Sigref*. The latter outputs the state space's transition system as a BDD expressed in Sigref's XML format.
Sigref	Two components were developed that specialise Sigref [Wim+06] to two cases, namely the case where a dynamic fault tree has to be transformed to its Markov model, i.e. *DFT to MRMC*, and the case where the state space interpreted as an IMC has to be transformed to a CTMC, i.e. *Sigref to MRMC*. Both employ Sigref's weak bisimulation minimisation algorithms on IMCs.

MRMC	This core engine is responsible for probabilistic model checking [Kat+11]. To overcome anticipated issues with numerical instability rising from stiffness in the failure rates, we enhanced MRMC with a Krylov-based method (c.f. Chapter 8) for computing transient probabilities.

All these building blocks together provide the core functionality of the COMPASS toolset. There are two interfaces with which the user can interface, namely the console-based interface and the graphical user interface. These interfaces invoke functionality of the building blocks and transform results back to an user-friendly representation. The two interfaces will be briefly discussed in respectively Section 5.3 and Section 5.4.

5.2 Extensions

Extension have been developed upon COMPASS, but did not (yet) become part of the COMPASS distribution. By our idea and under our supervision, these extensions were developed as Master's final projects during the development of COMPASS itself.

One extension does a translation to Promela instead of SMV. It was developed to explore the merits of using the SPIN model checker as a backend to SLIM. One of the advantages of the Promela translation is that it can handle reals in an explicit state way, whereas the SMV translation resorts to an encoding to SAT. See the Master's thesis [Ode10] for more details on this extension.

Slicing is the second extension. It takes a `CompileInfo` data-structure along with a set of properties of interest. By computing a smaller model that preserves validity of the properties of interest on the model, less verification resources is needed. The slicing algorithm does a static analysis on the data and control flow by accounting for the properties. Preliminary results indicate that the performance gain depends heavily on the properties. See the Master's thesis [Ode10] for more details on this extension.

Impact isolation is the third extension. It determines how transition effects have impact on future transitions by analysing the partial order relation in the global state space. This allows the user to determine to which degree SLIM transitions can affect others, such that a boundary scope of isolation is formally set up. It leads to an increasing understanding of coupling and cohesion, which benefits verification and validation efforts. The prototype implementation, written in Java, relies on the SLIM to Promela translator and a modified SPIN model checker, and then can visualise the range of impact of a single transition. See the Master's thesis [Ern12] for more details on this extension.

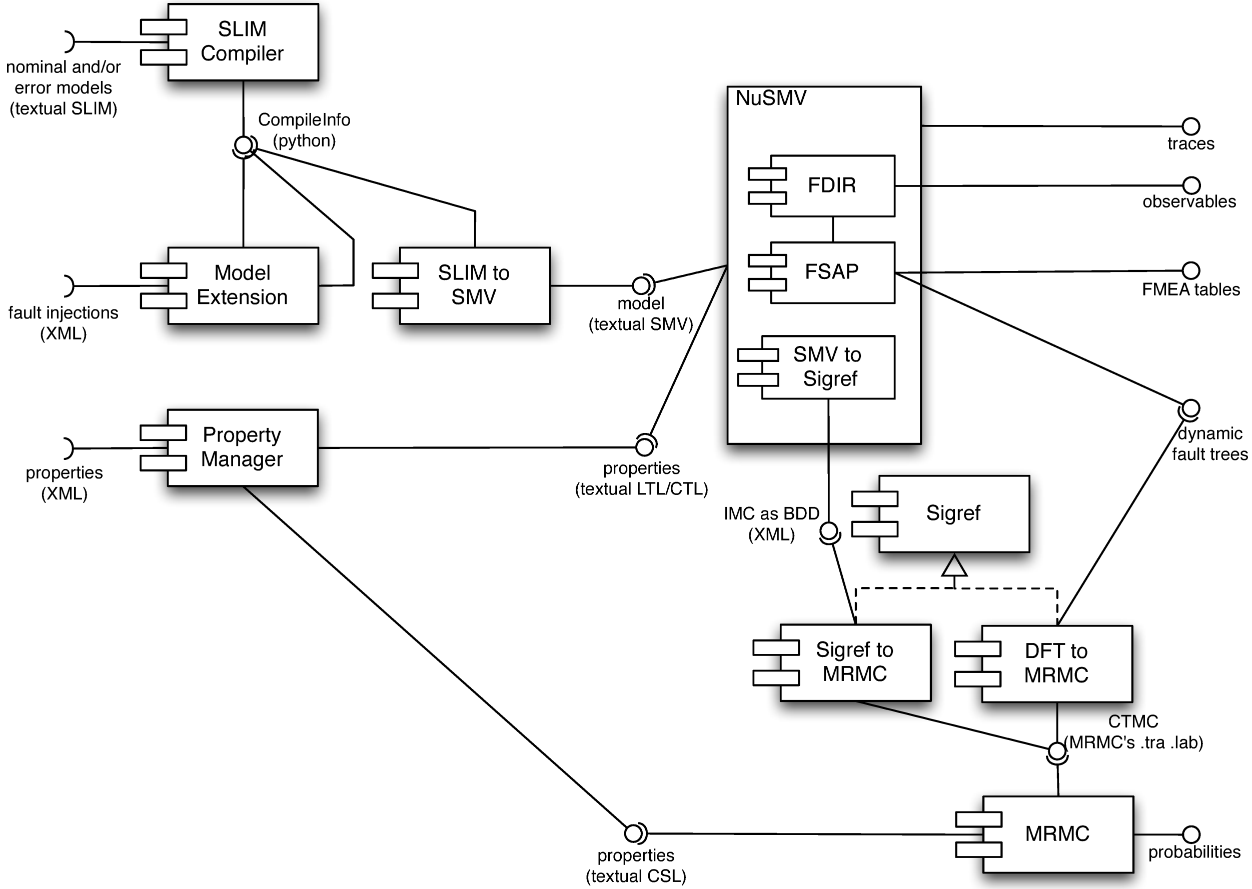
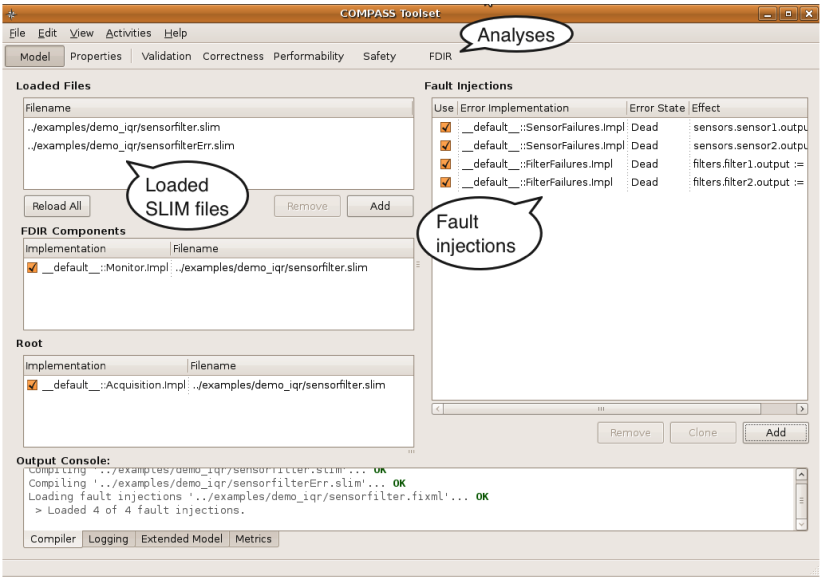

SLIM Compiler
nominal and/or error models (textual SLIM)
CompileInfo (python)
Model Extension
fault injections (XML)
SLIM to SMV
model (textual SMV)
Property Manager
properties (XML)
properties (textual LTL/CTL)
NuSMV
FDIR
FSAP
SMV to Sigref
traces
observables
FMEA tables
dynamic fault trees
IMC as BDD (XML)
Sigref
Sigref to MRMC
DFT to MRMC
CTMC (MRMC's .tra .lab)
MRMC
probabilities
properties (textual CSL)

Figure 5.1: UML Component diagram of the COMPASS toolset.


COMPASS Toolset
File Edit View Activities Help
Analyses
Model Properties Validation Correctness Performability Safety FDIR
Loaded Files
Reload All
Remove
Add
Loaded SLIM files
Fault injections
FDIR Components
Root
Fault Injections
Remove
Clone
Add
Output Console:
Compiler Logging Extended Model Metrics

Filename
../examples/demo_iqr/sensorfilter.slim
../examples/demo_iqr/sensorfilterErr.slim

Implementation	Filename
☑ __default__::Monitor.Impl	../examples/demo_iqr/sensorfilter.slim

Implementation	Filename
☑ __default__::Acquisition.Impl	../examples/demo_iqr/sensorfilter.slim

Use	Error Implementation	Error State	Effect
☑	__default__::SensorFailures.Impl	Dead	sensors.sensor1.outpu
☑	__default__::SensorFailures.Impl	Dead	sensors.sensor2.outpu
☑	__default__::FilterFailures.Impl	Dead	filters.filter1.output :=
☑	__default__::FilterFailures.Impl	Dead	filters.filter2.output :=

```
Compiling '../examples/demo_iqr/sensorfilter.slim'... OK
Compiling '../examples/demo_iqr/sensorfilterErr.slim'... OK
Loading fault injections '../examples/demo_iqr/sensorfilter.fixml'... OK
 > Loaded 4 of 4 fault injections.
```

Figure 5.2: Screenshot of the COMPASS's main window.

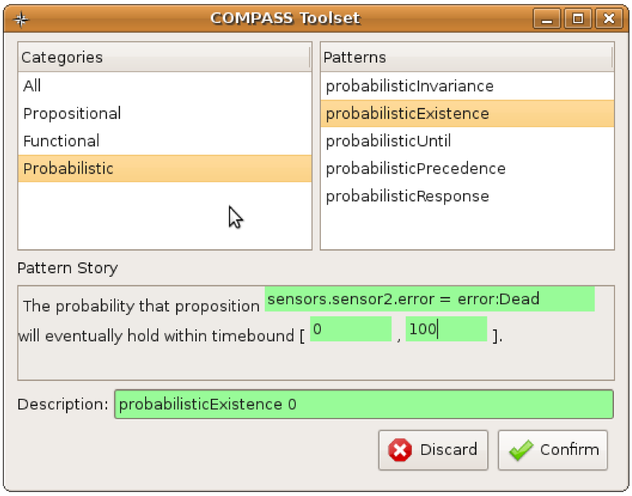
5.3 Console Interface

The console interface is designed to enable running analyses in (repeated) batches and allow for the integration of COMPASS functionality in other tool-chains. It is purely non-interactive and all inputs have to provided to the command-line. The COMPASS user manual [Bit+11b] provides examples and explanations on the console interface.

5.4 Graphical Interface

The graphical interface is designed to run analyses while developing the model. See Figure 5.2 for the main screen. It allows for single click reloading of the model. Furthermore, it provides user-friendly means for expressing fault injections and properties, by presenting them as on-the-fly validating forms. See Figure 4.1(a) for the form of fault injection and Figure 5.3 for the property entry form. The graphical interface can export fault injections and properties to its XML format, which can be used for the console interface to run the analyses.

The graphical interface is written in Python, GTK, PyGTK and the PyGTKMVC framework. It defines each function as a model, view and controller, and interrelated information between GTK models is updated through an event-subscription system.


COMPASS Toolset
Categories
All
Propositional
Functional
Probabilistic
Patterns
probabilisticInvariance
probabilisticExistence
probabilisticUntil
probabilisticPrecedence
probabilisticResponse
Discard
Confirm

Pattern Story

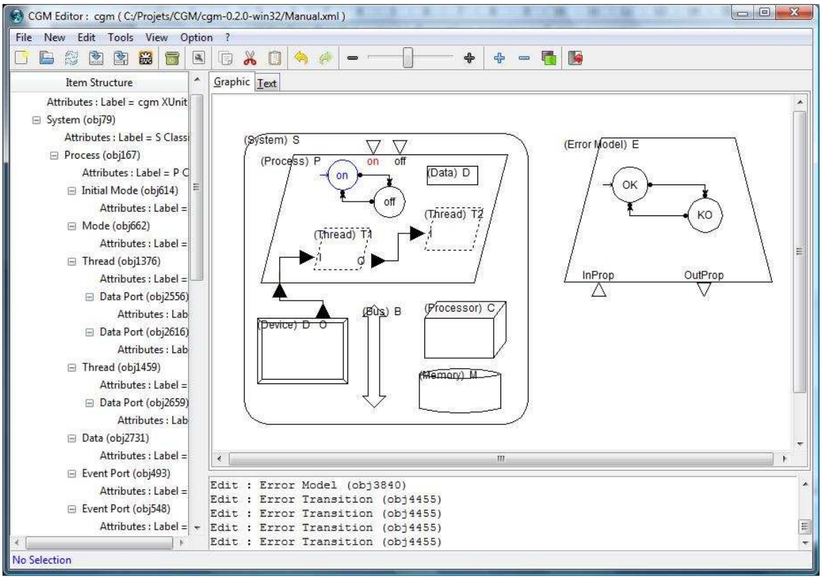
The probability that proposition sensors.sensor2.error = error:Dead will eventually hold within timebound [0 , 100].

Description: probabilisticExistence 0

Figure 5.3: Screenshot of the COMPASS's property form.

5.5 COMPASS Graphical Modeller

The COMPASS graphical modeller (i.e. CGM) provides a drag-and-drop graphical user interface (see Section 3.5) enabling the user to construct SLIM models graphically (see Figure 5.4 for an example of the graphical notation). The SLIM models constructed by the CGM are saved as a XML format called the SXML format. The CGM provides an export to SLIM function, such that the graphical SLIM models can be loaded into the COMPASS toolset. To ensure that exported SLIM files are valid, the SLIM compiler from the COMPASS toolset is enhanced to interact with CGM. Syntax violations detected by the SLIM compiler are pinpointed to a graphical SLIM element and the CGM will highlight those errors. Additionally, a function is added to import SLIM models that were not developed by the CGM. As they have no graphical layout, a rudimentary graph layouting algorithm is used to provide an initial layout that can be further refined by the user.


CGM Editor : cgm (C:/Projets/CGM/cgm-0.2.0-win32/Manual.xml)
File
New
Edit
Tools
View
Option
?
Item Structure
Attributes : Label = cgm XUnit
System (obj79)
Attributes : Label = S Classi
Process (obj167)
Attributes : Label = P C
Initial Mode (obj614)
Attributes : Label =
Mode (obj662)
Attributes : Label =
Thread (obj1376)
Attributes : Label =
Data Port (obj2556)
Attributes : Lab
Data Port (obj2616)
Attributes : Lab
Thread (obj1459)
Attributes : Label =
Data Port (obj2659)
Attributes : Lab
Data (obj2731)
Attributes : Label =
Event Port (obj493)
Attributes : Label =
Event Port (obj548)
Attributes : Label =
Graphic
Text
(System) S
(Process) P
on
off
on
off
(Data) D
(Thread) T2
(Thread) T1
I
O
I
(Device) D
O
(Bus) B
(Processor) C
(Memory) M
(Error Model) E
OK
KO
InProp
OutProp
No Selection

```
Edit : Error Model (obj3840)
Edit : Error Transition (obj4455)
Edit : Error Transition (obj4455)
Edit : Error Transition (obj4455)
Edit : Error Transition (obj4455)
```

Figure 5.4: Screenshot of the COMPASS graphical modeller.

6

Satellite Platform Case Study

Industrial case studies have been conducted earlier within the COMPASS project, such as a satellite's FDIR mode management system and a satellite's thermal regulation system [Boz+10a]. These studies were focussed on critical subsystems of a satellite, and did not cover the full functionality of a satellite platform such as the interaction between several subsystems. This chapter presents a newer and more comprehensive case study covering a satellite platform that is currently under development. Due to the confidential nature of the case, the model is not publicly available.

6.1 Case

At the highest conceptual level, the satellite is composed of the payload and the platform. The payload comprises mission-specific subsystems and the platform contains all subsystems needed to keep the satellite orbiting in space. The payload is usually designed and tailored from scratch, whereas for the platform lots of design heritage applies. For this reason, our case study focuses on a SLIM model of the platform, as this might benefit future projects too. A decomposition of the platform into selected subsystems is shown in Figure 6.1.

The majority of these subsystems are designed with degrees of fault-tolerance, depending on the criticality of the subsystem. Hot and cold redundancies with reconfigurations, voting algorithms, correcting codes and compensation procedures are part of comprehensive strategies for achieving fault-tolerance. In the extreme case, the satellite should survive a particular number of days without ground intervention assuming no additional failure occurs. As faults could occur at any level in the system's hierarchy (system, subsystem, equipment), the fault management system obeys a cross-cutting design according to the Fault Detection, Isolation and Recovery (FDIR) paradigm. This paradigm separates fault managements into three functions. The function of fault detection continuously monitors the system and

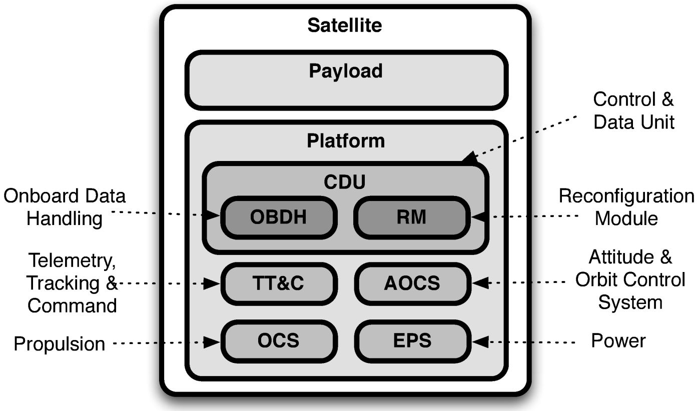

Satellite
Payload
Platform
CDU
OBDH
RM
TT&C
AOCS
OCS
EPS
Control & Data Unit
Onboard Data Handling
Reconfiguration Module
Telemetry, Tracking & Command
Attitude & Orbit Control System
Propulsion
Power

Figure 6.1: Decomposition of the case study's satellite. OCS consists of a series of thrusters for orbit corrections. AOCS is a control system consisting of several kinds of sensors for acquiring and maintaining a correct attitude and orbit. CDU is the main computer. EPS consists of solar arrays and batteries for satelite power. TT&C is the radio communication interface for ground control on Earth.

in case of anomalous values, emits appropriate events to react upon them. Monitoring is decentralised and performs at all levels of the system's hierarchy. After emitting fault detection events, fault isolation kicks in. This function is responsible for identifying the affected system's scope by determining the cause of the fault events. The function of fault recovery then takes appropriate actions to mitigate the fault events, and if possible, return to a nominal state.

As faults can occur at all levels, and that their effects can propagate throughout the system horizontally (same level) and vertically (across levels), the system is partitioned into five levels, in which the complexity of FDIR functions are organised:

Level 0	Failures are associated to a single unit and recovery can be performed by the unit itself.
Level 1	Failures are associated to single subsystem, and an external subsystem, the on-board software is responsible for its mitigation.
Level 2	Failures are associated to a multiple subsystems, and an external subsystem, the on-board software is responsible for its mitigation.
Level 3	Failures are occurring in the on-board software or in the processor modules. Dedicated reconfiguration modules are responsible for its mitigation.
Level 4	Failures that are not covered by lower level failures and are completely managed by hardware.

Failures are mitigated at its appropriate level. As with most Earth orbiting satellites, this satellite is required to be single-fault tolerant. If a fault is detected, all FDIR monitoring is ignored and the isolation and recovery of the detected fault is prioritised. As such, it can only handle one fault at a time.

6.2 Objectives

We started our case study at the preliminary design review stage (PDR) of the satellite project, where the details of design started to mature. In the traditional space engineering process, several objectives have to be met in order to proceed to the critical design stage, among which the following are of interest to us and thus within the scope of this case study:

- compliance of the preliminary design with the functional and operational requirements and justifications.
- demonstration of compliance with preliminary reliability, availability, failure tolerance, and failure propagation requirements
- consistency of HW/SW redundancies and FDIR concepts.
- evidence of tracking/implementation of preliminary RAMS recommendations.
- consistency and completeness of the preliminary RAMS analyses.
- completeness, credibility, and consistency of the preliminary design.

The satellite's development team was on a strict schedule, and hence it would be unwise to inject novel development approaches – like our initiative – into the production process. We ran our case study in parallel with the actual development as an experimental side-track. This benefits us, because we were not presented with fully crystallized and matured design documentations, but with volatile design information that was undergoing improvement and refinement. Findings in our case study were therefore directly relevant and could be provided as feedback to the satellite development team. Furthermore, we had to learn to cope with the constant influx of updated design details and how to continually adapt our own model to that. This experience, and the model itself, were our main objectives, which are identified as follows:

- *Reference model*: obtaining a formal model of a satellite platform as a reference for future formal modelling.
- *Toolset capability*: obtaining a model that pushes the limits of the COMPASS toolset and revealing directions for further research.

- *Modelling guidelines*: develop best practices for effective (e.g. fast) formal analysis using model checking techniques.
- *Improve software development life-cycle*: understand how formal analysis supplements and/or replaces existing practices as defined in ECSS (cf. Chapter 2. In particular, to understand the impact of increasing design maturity on formal modelling.

Note that for the latter, it is imperative that the case study is run in parallel with the system's active development. If our case study were run after its realization, the effort became an afterthought in which design information has fully crystallised and matured.

6.3 Modelling

The overall composite system is described by two important modes of satellite operation: *nominal* and *safe*. The nominal mode describes a set of satellite configurations in which the system functions within nominal conditions. Upon the detection of faults, recoveries might be attempted for resuming nominal operation. Otherwise a transition is made into safe mode for which the system reconfigures itself for survival until ground can perform an intervention. This important transition has system-level effects and hence is critical. In the remainder of this chapter this important event is called *TLE-1* (first top-level event).

During modelling, we focused on a subsystem/equipment at a time as the design of each corresponded more or less to a specific (section of a) design and a requirements document. We progressively increased coverage by adding more detailed subsystems to the overall model, while keeping high-level abstractions or stubs for the remaining subsystems. The metrics of the full model are described in the upper part of Table 6.1. Due to model confidentiality, we only highlight generalised modelling aspects and practices.

6.3.1 Discretisation

Various design aspects are often specified in terms of ranges. Like for the Sun sensors, ranges are used in degrees of Sun ray impact to determine exposure to the Sun. For the power system, over-currents are specified by voltage transfer functions. To avoid a combinatorial explosion of the state space, these ranges have to be abstracted with respect to the desired functionality, e.g. a Boolean indicating Sun exposure (or not) and respectively a Boolean indicating over-current (or not). Enumerations are used when there are gradations within the ranges.

Table 6.1: Metrics of the full satellite platform model and requirements.

Scope	Metric	Count
Model	Components	86
	Ports	937
	Modes	244
	Error models	20
	Recoveries	16
	Nominal state space	48421100
	LOC (without comments)	3831
Requirements	Propositional	25
	Absence	2
	Universality	1
	Response	14
	Probabilistic Invariance	1
	Probabilistic Existence	1

6.3.2 Timing

Real-time correctness is an important aspect for various subsystems, especially with regard to the recovery procedures. For example, the recovery modules contain a table of Programmable Alarm Patterns (PAP), which upon a match looks up a corresponding recovery procedure which is described by a Compressed Command Sequence (CCS). Each individual command in a recovery procedure is annotated with its maximum task duration. To enforce this timing behaviour in our model, a timer is added and transition guards over the timer are defined. If only guards were used, it would be possible for the system to stay in a mode forever (i.e. time divergence). To avoid this, mode invariants are added to force a transition when the invariant becomes invalid by the passage of time. A second concern with modelling timed aspects is to ensure the absence of Zeno behaviour. Otherwise the model could take infinitely many steps within a finite time-span. The presence of time-divergence and Zeno behaviour leads to invalid outputs, and hence their absence needs to be ensured. This is elaborated and discussed in Section 6.6.

6.3.3 Hybridity

Hybrid aspects (e.g. temperature evolution or fluid pressure) are a generalization of real-time constraints. They need to be incorporated into the model without discretisation if one wants to check compliance of range requirements, e.g., the temperature stays between a lower and upper limit in the presence of a (redundant) heating system. To ensure computational tractability [Aud+05], the COMPASS toolset only supports simple linear differential equations for time-dependent evolution. As many equations are not specified in this form, the engineer needs to abstract the original equations into linear ones. Additionally, the concerns of Zeno behaviour and time-divergence also apply in a similar fashion to timed models.

6.3.4 Reconfiguration

SLIM offers mode-dependent activations as a first-class language construct, allowing the enabling/disabling of components based on the current mode. Fault tolerance by redundancy can thus be easily expressed by modelling multiple components of equal functionality which are active in disjunct modes, like two processor modules being active in respectively the nominal and safe mode. Events from a recovery procedure can trigger transitions between the modes, resulting in a reconfiguration of the system topology.

6.3.5 Errors and Fault Injections

Our primary source for error modelling is the preliminary FMECA. It lists the possible detectable failures as an event and relates it to the effect on the system. This mapping is nearly equal to the fault injections. It also provides the information for constructing the error models. We found that in all cases, the probabilistic behaviour was either shaped as a single step from an error-free state to an error state (so-called permanent errors), or that they follow a fault-repair loop-structure on the error-free/error states (so-called transient errors). The FMECA is also the source for failure rates. They are expressed in failures in time (FIT), which indicates the expected number of failures in 10^9 hours.

6.3.6 Traceability

For any system under development, and especially in the preliminary design phase, the design is susceptible to changes. Every few months a new version of design documents is distributed with detailed change-logs. To keep track of the changes, we maintained a traceability spreadsheet that maps each SLIM part to the corresponding points in the design documents. Upon a new revision, we simply traversed the change-logs, pinpointed the affected parts in the SLIM model and updated them to reflect the change accordingly.

6.3.7 Assumptions

At first sight, the amount of design information is so overwhelming, that it is inconceivable to comprehend the system all at once, especially if one is not familiar with the system under development. Information might be perceived as incomplete, unclear, or wrong due to this, and this delays the modelling phase. We developed a practice of quickly continuing modelling using assumptive modelling decisions:

Abstraction	Describes how a SLIM element abstracts a part of the system.
Assumption	Describes how a SLIM element captures an assumptive understanding of the system.
Direct Conversion	Describes how a SLIM element maps directly to a part of the system.
Underspecified	Describes how parts of the system design documentation were insufficient for formal representation.
Explained	Are assumptions that have been clarified during review meetings.

During review meetings, we had the opportunity to check assumptions, and once these were clarified, the assumption was resolved (Explained).

6.4 Requirements Specification

Requirements documents are developed for all parts of the satellite at all levels (system, subsystem, equipment, etcetera). Furthermore, a particular set of requirements (e.g. system-level requirements) could function as a baseline for a set of lower-level requirements (e.g. subsystem requirements). Not all requirements were amenable for formal analysis. This has several reasons.

High-level requirements typically function as an umbrella for more detailed requirements. They typically lack the detail needed for formal verification. This can be seen in there form, which is typically prose-like, e.g. “FDIR functions must be active in all AOCS modes”.

A significant part of the requirements do not only cover behaviours, but also reflect the system’s organisation. They state which components should be present, and they state how a component is structured in subcomponents, and which components may communicate with each other. They give rise to the system hierarchy. As these a static in our modelling language, verification of them is out of the scope of our activities.

In early design, often requirements are specified about the existence of a behaviour, without describing how to realise this behaviour. An example is that FDIR behaviour should be present to detect, isolate and mitigate faults, but it is left unspecified how to achieve this. These kind of requirements are typically subject to refinement is detailed design, where decisions are made on the exact required behaviour. These requirements are typical intended underspecification.

Also, unintended underspecification of requirements might occur. These are not trivially perceived as the requirements specifications we used do not explicitly mark requirements as intentionally underspecified. Without an extensive background in satellite engineering, and as such a clear picture of unintended underspecification, it is difficult to spot them.

In other cases, requirements can describe behaviours that are out of scope of our objectives. These requirements could for example cover behaviours that are intentionally abstracted away in the model. The behaviour of the AOCS control loop is an example which is typically expressed in terms of transformations of sensor-data (e.g. Sun light angles, Earth sensor intensities). Such behaviour is typically abstracted to the status of the behaviour, e.g. whether it is nominal, degraded, or failed. Also, requirements could also refer to the payload, and this was intentionally left out the scope of our objectives.

From the several thousands of requirements we obtained, we analysed 106 requirements and checked them using the above criteria. From the 106 requirements, 24 were suitable and used for our model. Once deemed suitable, they had to be mapped to a specification pattern. In many cases, clarifications are needed during the mapping. For example, which analysis (and why) is suitable for verification? What are the applicable modelled components? What constitutes the atomic propositions (e.g. what is exactly a FDIR function)? Hence for the analysis and mapping of requirements, we additionally maintained an assumptions spreadsheet and a traceability spreadsheet, just like we did for modelling. Similar to the studies in [DAC99; Gru08], we tracked the kind of patterns (cf. Section 4.2) used and these are shown in the lower part of Table 6.1.

6.5 Analyses

Modelling is highly intertwined with analysis, since the output from analysis provides valuable information for possible refinements of the model. The most widely-used analysis method during modelling is model simulation, as inspection of traces is a fast sanity check before running a resource-consuming analysis.

During all analyses, particular sets of fault injections were disabled/enabled depending on the aim. This was needed for this case study as we observed that fault injections lead to a significant increase of the state space (see Figure 6.2). This is not surprising. A fault injection basically yields the cross-product of the subsystem to which the error is injected, and the error model. There is no direct correlation between the amount of fault injections and the increase, although there is a relation between the kind of fault injections and the increase. Fault injections that have system-level impact (e.g. processor module failures) add more behaviour than fault injections with lower-level impact (e.g. Earth sensor failures) as they affect a larger fragment of the state space.

All analyses were run on a set of identical computers running 64-bits Linux, each with a 2.1 GHz AMD Opteron CPU and 192 GB RAM. The consumption of peak resources for each analysis is shown in Table 6.3.

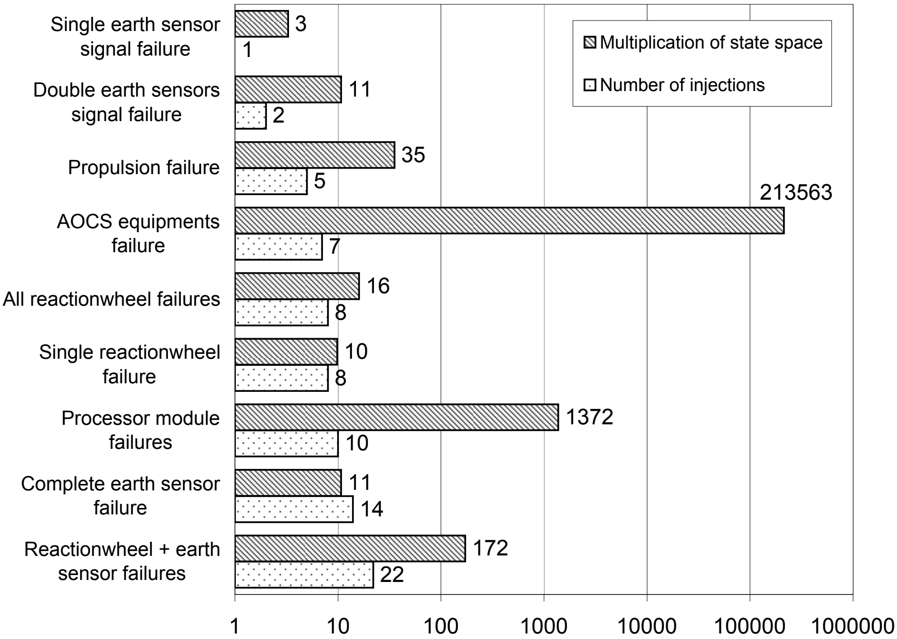


Figure 6.2: Degrees of state space increase with respect to nominal state space size when injecting failures. The scale is logarithmic.

6.5.1 Functional Verification

We separated this into two activities: discrete and real-time/hybrid verification. During the verification of the discrete part, which is the majority of the model, we verified 16 properties. Noteworthy here is that the COMPASS toolset does verification in the absence of any fairness constraints. Occasionally, it would be useful to express those to avoid starvation of components. Now we had to embed the constraints in the model instead, which slightly increases its size due to the added synchronization.

The real-time/hybrid parts of the model are relatively small and were joined together into a single hybrid model. This alternative model was developed to check a requirement stating that the redundant heater is only active in degraded operations. Verifying this requires the bounded model checking backend [Aud+02] and we experimented with increasing bounds to measure the limit (cf. Table 6.3). With respect to the increasing bounds, we measured that the time needed grows exponentially and memory-wise the growth is linear. We stopped our measurements at bound 79, as this exhausted our machine.

6.5.2 Safety & Dependability

The platform's most critical event that affects safety and dependability is TLE-1, i.e. the transition to safe mode. The transition is triggered upon the occurrence of severe failures. In the design documents, a (static) fault tree of 66 nodes is provided relating TLE-1 with the failures. Using our toolset, we could produce the same (static) fault tree from our SLIM model in a *fully automated* manner. We also generated a fault tree for the setting of the fail-operational flag. This flag indicates that the satellite's payload services might be impaired due to platform failures. The dynamic variant of fault tree analysis delivered similar results. In two cases, it delivered them with less computation time. This came to us as a surprise, given that the dynamic aspect is an additional analysis upon static fault tree generation. The internal logs of the COMPASS toolset revealed this is due to its implementation, and likely due to dynamic BDD variable reordering which on our model is more favourable for dynamic fault tree generation. A FMEA table was generated for mapping the sensor failures with three system effects: detection of failures, the setting of the fail-operational flag and TLE-1. The generated table did not provide additional values on the fault tree, as it directly maps failures to the user-provided effects. It would be more interesting if the COMPASS toolset could synthesize a mapping from failures to a chain of effects, showing how the first effect directly caused by the failure propagates through the system to subsequent effects and eventually becoming a failure like TLE-1.

6.5.3 Fault Management Effectiveness

For fault detection, we checked which observables were triggered when the transition to safe mode is made. This could trigger 129 observables. Subsequently, fault isolation was performed on all 129 observables. No properties are used for this, since the observables themselves are the only required inputs. Diagnosability analysis was performed to see whether a double Earth sensor failure is diagnosable (for the satellite operator) when TLE-1 occurs. Without any result, we had to stop the analysis after 7 days and consuming nearly 1400 MB at its peak. This is understandable as, contrary to model checking algorithms which usually compute a single state space, diagnosability performs a coupled reachability analysis (cf. Section 4.10).

6.5.4 Performability

Reliability requirements are usually defined as a cumulative distribution function and state that the foreseen reliability must be at least as good. Its probabilistic nature fits performability analysis. On our model, we wanted to determine the reliability of the satellite in the presence of a sensor failure. Performability analysis however ran out of allocatable memory after nine hours. Investigation revealed that the transformation of the state space into its underlying Markov chain ran out of allocatable memory. The transformation involves the use of a weak bisimulation minimization algorithm, whose implementation in COMPASS is an adapted version of Sigref [Wim+06]. During our case study, we observed that that implementation could only allocate memory up to two GB, hence the out of memory.

Another approach to verifying the reliability requirements is by computing the probabilities of the TLE-1 fault tree which was generated during safety & dependability analysis. This is called fault tree evaluation. As shown in Table 6.3, the computation occurs in a split second.

Note that though both approaches can be used for this requirement, there are substantial differences. Fault trees are essentially abstract state spaces where the relations between the top-level-events and the failures are conservatively over-approximated by AND-, OR- and PAND-gates (Priority AND). With performability on the other hand, these relations are precisely preserved, which however comes with increased complexity when the underlying Markov chain needs to be obtained. This is discussed further in Section 4.12.

6.6 Discussion

In this section, we reflect our objectives as stated in Section 6.2 and discuss the outcomes of this industrial case study.

6.6.1 To PDR Objectives

For entering the subsequent stage in the development process, the Preliminary Design Review (PDR) objectives have to be met. During our efforts, we encountered several inconsistencies in the design documents. Most of them were found during modelling, due to the critical interpretation of the design documents. They were reported and have been corrected.

6.6.2 Reference Model

This SLIM model is the largest and most-comprehensive we developed to date that is suitable for model checking. Its incorporation of probabilistic aspects through errors, and real-time/hybrid aspects have made it a reference for benchmarking new algorithms that underlie the analyses. Additionally, it can be used for kick-starting subsequent formal modelling activities, so that one does not have to start modelling from scratch.

6.6.3 Toolset Capability

As reported in an earlier evaluation of much smaller scale [Boz+10a], the hierarchical and component-oriented nature of the modelling language fits naturally a development by refinement process. During this case study of much broader scope

Table 6.3: Peak computation times and memory usage of the analyses.

Analysis	Fault Injections	Properties	Time (sec)	Mem. (MB)
Discrete model checking	(none, i.e. nominal behavior only)	“health check on valves is performed” and “no firing of thrusters triggers reconfiguration” and “thrusters not stopping firing triggers reconfiguration” and “overpressure triggers opening latch valve”	224	122
Discrete model checking	Single Earth sensor signal failure	i.d.	296	125
Discrete model checking	Double Earth sensors signal failure	i.d.	677	132
Hybrid model checking (10^*)	Single Earth sensor signal failure	“No thruster usage during nominal operation”	23	242
Hybrid model checking (20^*)	Single Earth sensor signal failure	i.d.	52	360
Hybrid model checking (30^*)	Single Earth sensor signal failure	i.d.	101	492
Hybrid model checking (40^*)	Single Earth sensor signal failure	i.d.	204	612
Hybrid model checking (50^*)	Single Earth sensor signal failure	i.d.	361	713
Hybrid model checking (60^*)	Single Earth sensor signal failure	i.d.	967	884
Hybrid model checking (70^*)	Single Earth sensor signal failure	i.d.	2176	1006
Fault tree analysis	Double Earth sensors signal failure	TLE-1	555	134
Fault tree analysis	AOCS equipments failure	TLE-1	2898	181
Fault tree analysis	Double Earth sensors signal failure	“fail-operational flag is set”	769	132
Fault tree analysis	Processor module failures	“CDU alarms are raised”	483	134

. . .
(continued on next page)

. . .
(continued from previous page)

Analysis	Fault Injections	Properties	Time (sec)	Mem. (MB)
Fault tree analysis	AOCS equipments failure	"fail-operational flag is set"	8349	239
Dynamic fault tree analysis	Double Earth sensors signal failure	"fail-operational flag is set"	630	135
Dynamic fault tree analysis	Processor module failures	"CDU alarms are raised"	547	136
Dynamic fault tree analysis	AOCS equipments failure	"fail-operational flag is set"	5581	212
FMEA table generation	Double Earth sensor signal failure	"failures are detected" and "fail-operational flag is set" and TLE-1	1003	134
Fault detection analysis	Double Earth sensor signal failure	TLE-1	1173	142
Fault isolation analysis	Double Earth sensor signal failure	n.a.[¶]	21920	136
Diagnosability analysis	Double Earth sensor failure	TLE-1	586093[†]	1474[†]
Performability	Single Earth sensor signal failure	TLE-1	33166[‡]	2103[‡]
Fault tree evaluation	Double Earth sensor signal failure	"fail-operational flag is set"	1	n.a.[§]
Dynamic fault tree evaluation	Double Earth sensor signal failure	"fail-operational flag is set"	1	n.a.[§]

[*] Bound parameter used in the bounded model checking.

[†] Ran out of time.

[‡] Ran out of memory.

[§] Analysis terminated too quickly for measurement.

[¶] Fault isolation requires only the model as an input.

and size, we highlighted additional points on the offered modelling constructs. We recognized a need to support flows on continuous variables, used for the hybrid aspects. This would allow for exposing its continuous evolution to its neighbouring components. In the same line, it would be useful to develop efficient algorithms for verifying systems with (decidable fragments of) non-linear equations, allowing for more fine-grained hybrid behaviour. Additionally, we encountered Zeno behaviour and time divergence several times (cf. Section 4.4), and found it difficult to manually pinpoint them in the model. The algorithmic detection of Zeno behaviour is an active field of research, and once it matures, it is desirable to have it included.

Regarding the COMPASS toolset itself, it is pleasant not to be exposed to the underlying logic and model checking tools. For most analyses, the performance and the features are sufficient. Other analyses are subject to improvement. Upon model checking for example, the ability of expressing fairness constraints for the absence of starvation is a more elegant way than expressing them in the model itself. For certain temporal logics such as LTL, it is possible to encode a rich class of fairness assumptions in the requirements. Regarding FMEA, we found that FMEA generation is making a reverse mapping of fault tree generation, hence not complementing the information provided by fault trees. What would be more useful is to understand the chain of effects (i.e. fault propagation) that start by a failure. This would give more information on their detection means and possibly the design of the recovery procedures. Regarding performability analysis, this analysis ran out of memory after nine hours (cf. Table 6.3). The cause is the weak bisimulation minimization implementation [Her02] used to transform the state space to its underlying Markov chain. Improvements to that implementation will have direct benefits to performability analysis. Diagnosability analysis on the other hand ran out of time. This is caused by the coupled reachability algorithm that underlies diagnosability (cf. Section 4.10). Faster model checking algorithms will improve its performance. Especially ones that exploit the compositional structure of the SLIM model, like our approach described in Chapter 7.

6.6.4 Modelling Guidelines

We used the preliminary FMECA, the requirements and the design documents for respectively the error models, properties and the model itself. Note that the inputs were in a rough state: they change due to review. Additionally, formal modelling and analysis supports the review process by forcing one to consider underspecification. Updates due to review can be nicely accommodated by exploiting SLIM's features for modelling by refinement. When design information is unclear, assumptions can be modelled which we captured along with the traceability of the modelled elements. In later phases, the assumptions can be checked or raised during review meetings as discussion points.

Regarding proper abstraction, it is wise to consider abstraction depending on the requirement that needs to be verified. For the majority of the cases, discretisation by enums and booleans are the natural way for abstraction. This is a necessity for keeping the growth of the state space under control. Only when real-time and hybrid aspects need to be verified alternative SLIM component implementations can be developed that incorporate such behaviour using the same component interfaces.

Careful attention has to be paid when modelling real-time and hybrid systems. We encountered Zeno behaviour and time-divergence several times. To avoid them, one can perform a manual inspection on the model, as explained in Section 4.4. These two manual checks are tedious, especially given a large model, but are needed as long as algorithmic detection of them is impractical.

Furthermore, during the case study, we developed a small number of architectural patterns for modelling frequently occurring concepts, like recovery procedures and particular redundant configurations. The patterns are now tailored to this case study, but could be further developed to become more generic.

6.6.5 Improving Software Development Life-cycle

It is generally understood that formal modelling and analysis provides outputs that improve the eventual system under development. Formal methods forces engineers to consider design issues early, and have them resolved long before integration testing, thus avoiding increased costs. In our case study, we detected several inconsistencies and reported them to the satellite development team. Although the benefits are clear, it is yet unclear how formal methods should be leveraged. There are currently no standardised guidelines on the use of formal methods in the software development life-cycle. For avionics systems, this situation changes with the third revision of European-American standard for software considerations in airborne systems, called DO-178C/ED-12C [IE11]. It incorporates guidelines and allows for creditation when formal methods are used for the development of avionics software.

The European space software development life-cycle (cf. Section 2.2) does not (yet) reflect the use of formal methods. Based on our experience of this case study, we think it is more pragmatic for the current E-40 standard to add aspects of formal modelling and analysis. Most importantly, to have a means to keep track with the evolution of design artefacts and ensure that the formal model reflects the current design. For this reason, we developed a simple but useful habit of keeping assumption spreadsheets and traceability tables (cf. Section 6.3). Assumption spreadsheets allowed us to progress swiftly on modelling, even when the details are unclear. Traceability tables allowed us to pinpoint, upon design changes, the affected parts of the model and push the changes to the model accordingly. These lessons are the outcome of running our case study in parallel with a system in active development, because otherwise we would have been presented a fully detailed and mature design in which all issues have been already resolved, and as such, we would not be forced to keep up with the changes.

7

Craig Interpolation-Based Compositional Reasoning

Curbing the state space explosion is one of the greatest challenges in model checking. For decades, researchers have studied compositional reasoning as the solution to this, resulting in a plethora of approaches that usually either reduce peak memory consumption, work only for loosely-coupled programs, or cannot be automated yet.

In this chapter, we present a proof-theoretic approach that aims to aggressively abstract sets of interacting components as environments to other components. The abstracted environment, which is smaller due to its smaller interaction alphabet, together with the component it interacts with, is obtained through processing the verification proofs of bounded model checking runs. The abstraction derived from it however are sound for unbounded verification. We argue that this technique is particularly effective for continuous reverification of models with large underlying state spaces.

7.1 Preliminaries

The result in this chapter builds upon existing work in satisfiability theory, symbolic model checking and interpolation. These topics are addressed in the following subsections.

7.1.1 Propositional Satisfiability

The propositional satisfiability problem is that given a Boolean formula, its variables can be assigned in such a way to make the formula evaluate to $\top$ (i.e. true). If this is the case, the formula is satisfiable, otherwise the formula is unsatisfiable. For example, the formula $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_3)$ is satisfiable with the valuation

σ, consisting of $\sigma(x_1) = \top$, $\sigma(x_2) = \top$ and $\sigma(x_3) = \top$. In such a formula, *literals* are variables (e.g. x_3) or a negation of the variables (e.g. $\neg x_3$). A *clause* is a disjunction of literals (e.g. $(x_1 \vee x_2 \vee \neg x_3)$). Traditionally, only the Boolean operators AND (i.e. $\wedge$) and OR (i.e. $\vee$) are used, and other Boolean operations like exclusive OR are expressed using $\wedge$ and $\vee$.

The worst-case time complexity of the propositional satisfiability problem is NP-complete, a classic result by Cook [Coo71]. This has not discouraged researchers to devise algorithms and heuristics to solve propositional satisfiability problems. Early algorithms did not scale well. As of 2002, yearly competitions emerged (cf. `satcompetition.org`, which highly stimulated this field, progressing to modern SAT-solvers capable of deciding (un)satisfiability of millions of clauses in mere seconds. The most modern ones even exploit multi-core systems in which benchmarks have shown a scalable reduction in computing time [WHM09].

Even though the satisfiability problem is about deciding satisfiability, SAT-solvers typically generate a proof of the outcome. In case a formula is satisfiable, it can generate a *satisfying assignment*, which are valuations to the occurring variables. Depending on the formula, it is possible that there are multiple satisfying assignments. In case a formula is unsatisfiable, SAT-solvers are capable of generating *unsatisfiable cores* as a proof. An unsatisfiable core is an unsatisfiable subformula of the original Boolean formula. An unsatisfiable core however does not contain the reasoning steps that show why the subformula is unsatisfiable, it merely makes the original problem smaller. Another type that does preserve the reasoning steps of the unsatisfiability proof is the *resolution refutation graph*, usually referred to by Π. It is formalised as a directed acyclic graph $G_\Pi = (V_\Pi, E_\Pi)$, where V_Π is a set of clauses (not necessarily a subset of the original formula). If a vertex $v \in V_\Pi$ is a root (there are usually multiple), then it is a clause in the original formula. Otherwise the vertex has exactly two predecessors, v_1 and v_2 of the form $v_1 \equiv x \vee D$ and $v_2 \equiv \neg x \vee D'$. The clause v is the simplification of $D \vee D'$ and x is its *pivot variable*. There is only one leaf which is the empty clause $\bot$. An example is shown in Figure 7.1. The resolution graph reasons how clauses, starting from the root clauses, have pivot variables that can be eliminated, as they contribute to the inconsistency. Once all variables are eliminated, the empty clause $\bot$ is reached, indicating unsatisfiability. Later in this chapter, the resolution graph is used to derive interpolants.

7.1.2 Symbolic Model Checking

Boolean formulae as described in Section 7.1.1 can be used to perform symbolic model checking. Intuitively, this approach of model checking does not explicitly enumerate all states, but traverses the state space by describing sets of states in a symbolic manner. The main ingredients are the initial condition and the transition function.

The *initial condition*, usually denoted as $I : \bar{s} \to \{\top, \bot\}$, is a Boolean formula represented as a propositional function (a.k.a. switching function) consisting of

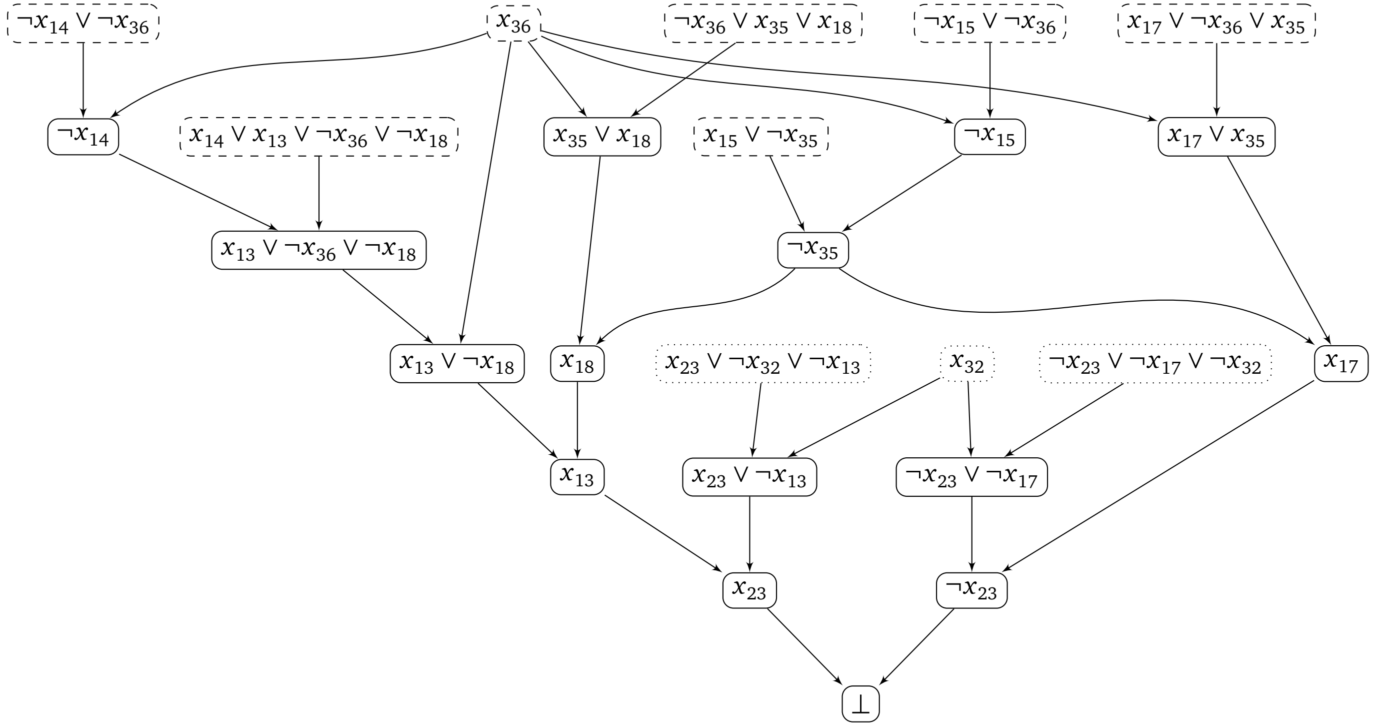

$\neg x_{14} \vee \neg x_{36}$
x_{36}
$\neg x_{36} \vee x_{35} \vee x_{18}$
$\neg x_{15} \vee \neg x_{36}$
$x_{17} \vee \neg x_{36} \vee x_{35}$
$\neg x_{14}$
$x_{14} \vee x_{13} \vee \neg x_{36} \vee \neg x_{18}$
$x_{35} \vee x_{18}$
$x_{15} \vee \neg x_{35}$
$\neg x_{15}$
$x_{17} \vee x_{35}$
$x_{13} \vee \neg x_{36} \vee \neg x_{18}$
$\neg x_{35}$
$x_{13} \vee \neg x_{18}$
x_{18}
$x_{23} \vee \neg x_{32} \vee \neg x_{13}$
x_{32}
$\neg x_{23} \vee \neg x_{17} \vee \neg x_{32}$
x_{17}
x_{13}
$x_{23} \vee \neg x_{13}$
$\neg x_{23} \vee \neg x_{17}$
x_{23}
$\neg x_{23}$
$\perp$

Figure 7.1: Example of a resolution refutation graph. The dotted and dashed nodes are roots occurring in respectively the A part and the B part of the formula.

Boolean variables $\bar{s} = s_1, \ldots, s_n$. Whenever a particular valuation σ of $\bar{s}$ holds by I, i.e. $\sigma(I) = \top$, then σ is an initial state. Multiple distinct initial states may hold by I.

The *transition function*, usually denoted as $T : \bar{s} \times \bar{s}' \to \{\top, \bot\}$, is a propositional function with $\bar{s} = s_1, \ldots, s_n$ and $\bar{s}' = s_1', \ldots, s_n'$. Note that the cardinalities of $\bar{s}$ and $\bar{s}'$ are equal. If for a pair of valuations σ and σ' the transition functions holds, i.e. $\sigma\sigma'(T) = \top$, then σ' is a valid successor state to state σ.

The initial condition and the transition function are used to compute reachable states. For example, the Boolean formula resulting from $I \wedge T$ are all the states reachable in one step from the initial state. Those states are captured by the primed variables $\bar{s}'$. To compute the subsequent successor states, we first need to substitute the variables $\bar{s}'$ to $\bar{s}$, using the substitution operator: $(I \wedge T)[\bar{s}'/\bar{s}]$. The resulting formula can be conjuncted with T to determine the two-step reachable states. By repeatedly performing this operation, and accumulating the reachable states, the full reachable state space can be computed.

A particular approach to symbolic model checking is by using SAT-solvers. This is called *bounded model checking*, since the early approaches [Bie+99] required the user to provide a bound $k \in \mathbb{N}$. The bound is used to *unroll* the transition condition such that the resulting formula represents all states reachable within k steps: $I[\bar{s}/\bar{r}_0] \wedge T[\bar{s}/\bar{r}_0, \bar{s}'/\bar{r}_1] \wedge \ldots \wedge T[\bar{s}/\bar{r}_{k-1}, \bar{s}'/\bar{r}_k]$. To ease our notation, the aforementioned formula will subsequently be referred to as:

$$I_0 \wedge T_1 \wedge \ldots \wedge T_k \tag{7.1}$$

The bound is necessary for tractability of the SAT-solver, otherwise the formula would unroll beyond its capabilities. In the above formula, each $\bar{r}_i$, with $0 \leq i \leq k$, captures the states reachable in i steps. To verify an invariant ϕ, the formula can be conjuncted with $\bigvee_{i=0}^{k} \neg\phi[\bar{s}/\bar{r}_i]$, the invariant negated. To ease our notation, each $\neg\phi[\bar{s}/\bar{r}_i]$ shall be referred to as $\neg\phi_i$. If the SAT-solver finds the formula satisfiable, the property does not hold within k steps, i.e. $M \not\models^k \phi$. The satisfying assignment is a counterexample. If the SAT-solver proves the formula is unsatisfiable, then it only means the property holds up to depth k, i.e., $M \models^k \phi$, and an outcome w.r.t. the full state space remains inconclusive.

7.1.3 Interpolation

The cornerstone of our compositional reasoning is a classical result by William Craig, his interpolation theorem for first-order logic [Cra57]:

Theorem 7.1.1 (Craig's Interpolation Theorem). *Let A and B be formulae over first-order logic. If $A \implies B$ holds, then there exists an interpolant C expressed using the common variables of A and B such that $A \implies C$ and $C \implies B$ holds.*

Proof. We describe here the proof approach by [Bus97]. It is a proof for the propositional case, which is sufficient for our purposes. Let $\bar{x}$ be the variables in A

which are uncommon with B, $\bar{y}$ be the common variables of A and B and $\bar{z}$ be the variables of B uncommon with A. Let $|\bar{y}|$ be k. Let $\sigma_1 \dots \sigma_m$ be all satisfiable assignments to $\bar{y}$ such that A holds by further assignment of values to $\bar{x}$. We can construct the interpolant C from $\sigma_1 \dots \sigma_m$ by an exponential construction:

$$C = \bigvee_{i=1}^{m} (y_1^{(i)} \wedge \ldots \wedge y_k^{(i)})$$

where

$$y_j^{(i)} = \begin{cases} y_j & \text{if} \quad \sigma_i(y_j) = \top \\ \neg y_j & \text{if} \quad \sigma_i(y_j) = \bot \end{cases}$$

This construction shows that $A \implies C$ holds. On the other hand, a satisfying assignment to $\bar{y}$ can be extended to a satisfying assignment to $\bar{y}, \bar{x}$ that satisfies A. Since $A \implies B$, every extension of this satisfying assignment to $\bar{y}, \bar{x}, \bar{z}$ must satisfy B. Therefore $C \implies B$. □

The beauty of Craig's interpolation theorem is that C is expressed using a subset of variables occurring in A, showing that the validity of $A \wedge B$ depends only on the variables common between A and B. We shall later on use this property to formulate our component-oriented interpolation theorem.

SAT-solvers take formulae in conjuncted normal form as input, and the interpolation theorem can be reformulated in such a way that it applies to conjunction of clauses. First observe that $A \implies B$ is equivalent to $\neg(A \wedge \neg B)$. This means that a tautology of $A \implies B$ is equal to a contradiction of $A \wedge \neg B$. By Craig's interpolation theorem it follows that if $A \wedge \neg B$ is unsatisfiable, there exists an interpolant C such that $A \implies C$ holds and $C \wedge \neg B$ is unsatisfiable. In this shape, the unsatisfiability of a formula indicates the existence of an interpolant.

The interpolation theorem only postulates the existence of an interpolant, and its proof shows how to construct one that is exponentially sized in the number of common variables. The proof assumes that all the satisfying assignments to A are known. We will use another approach by exploiting the ability of some SAT-solvers to generate resolution refutation proofs, from which interpolants can be derived [McM05]. This is demonstrated in the following definition:

Definition 7.1.2 (Interpolant Construction from Resolution Refutation Graph)**.** Let Π be the resolution refutation for $A(\bar{x}, \bar{y}) \wedge \neg B(\bar{y}, \bar{z})$ with the graph $G_\Pi = (V_\Pi, E_\Pi)$ associated with it. For each vertex $v \in V_\Pi$, let v_1 and v_2 be its predecessors and let

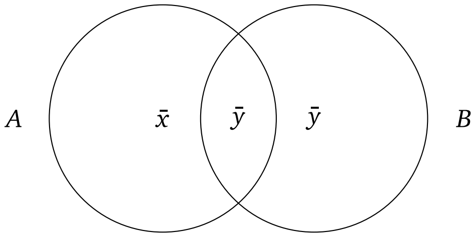


Figure 7.2: Relation of sets of variables occurring in $A(\bar{x},\bar{y}) \wedge \neg B(\bar{y},\bar{z})$.

C_v be a Boolean formula such that

$$\begin{aligned}
\bot &\iff v \in A \text{ and } v \text{ is root} \\
\top &\iff v \in B \text{ and } v \text{ is root} \\
(\neg y_i \wedge C_{v_1}) \vee (y_i \wedge C_{v_2}) &\iff y_i \in \bar{y} \text{ and } y_i \text{ is the pivot variable of } v \\
& \text{ and } y_i \in v_1 \text{ and } \neg y_i \in v_2 \\
& \text{ and } v \text{ is non-root} \\
C_{v_1} \vee C_{v_2} &\iff x_i \in \bar{x} \text{ and } x_i \text{ is the pivot variable of } v \\
& \text{ and } v \text{ is non-root} \\
C_{v_1} \wedge C_{v_2} &\iff z_i \in \bar{z} \text{ and } z_i \text{ is the pivot variable of } v \\
& \text{ and } v \text{ is non-root}
\end{aligned}$$

This definition is complete with respect to the variables occurring in A and B. This is easily shown by the Venn diagram in Figure 7.2. If Definition 7.1.2 is applied from the leaf $\bot$, one gets an interpolant for A, which is formulated by the following theorem:

Theorem 7.1.3 (Correctness of Interpolant Construction from Resolution Refutation Graph). *Let $A \wedge \neg B$ be unsatisfiable with resolution refutation Π as the proof. Then $C_\bot$ is an interpolant for $A \wedge \neg B$.*

Proof. There are several versions to proof this theorem. We prefer the approach and notation described in [Hel07]. □

Example 7.1.4 (Interpolation from Resolution Refutation Graph). Consider Figure 7.1, which is a resolution refutation graph for formula $A \wedge \neg B$. Notice that the dotted vertices are part of A and the dashed vertices are part of $\neg B$. By applying Definition 7.1.2, one obtains the interpolant $\neg x_{13} \vee \neg x_{17}$. Note that $x_{13}, x_{14}, x_{15}, x_{17}$ are shared between A and B. The variables x_{23}, x_{32} occur strictly in A. The variables x_{18}, x_{35}, x_{36} occur strictly in B.

7.2 Component-Oriented Interpolation

In traditional model checking approaches of concurrent systems, the concurrent processes are composed using a parallel composition operator. The result is a single global transition relation that represents the concurrent system. In our approach, we leverage the existing composition of processes to perform a compositional reasoning approach. For multi-threaded programs, the composition is typically asynchronous, whereas for hardware circuits, the composition is typically synchronous. In the upcoming, we will describe our approach on the synchronous case only. It can however be extended to the asynchronous case by casting the asynchronous model into a synchronous one.

A synchronous composition of n processes $M_1, \ldots, M_n$, with their associated transition relations $T^1, \ldots, T^n$, is $T = \bigwedge_{i=1}^{n} T^i$. The associated initial conditions are $I = I^1 \wedge \ldots \wedge I^n$. When this is applied to the bounded model checking formula, see equation (7.1), the result is the following:

$$\bigwedge_{i=1}^{n} I^i \wedge \bigwedge_{i=1}^{n} T_1^i \wedge \ldots \wedge \bigwedge_{i=1}^{n} T_k^i \wedge (\neg\phi_0 \vee \ldots \vee \neg\phi_k)$$

Let us now isolate a particular process M_p, such that it becomes more apparent how A and $\neg B$ are to be determined:

$$\underbrace{I^p \wedge T_1^p \wedge \ldots \wedge T_k^p \wedge (\neg\phi_0 \vee \ldots \vee \neg\phi_k)}_{\neg B} \wedge \underbrace{I^{\neq p} \wedge T_1^{\neq p} \wedge \ldots \wedge T_k^{\neq p}}_{A} \tag{7.2}$$

In the above, $T_i^{\neq p}$ is defined as $\bigwedge_{q \in 1 \ldots n \setminus p} T_i^q$. From Theorem 7.1.1, it follows that whenever ϕ holds within bound k, there exists an interpolant C, such that it is implied by A. Intuitively, the interpolant can be perceived as the transition relation representing the k-bounded environment of process p. The interpolant is however significantly smaller than the original formula representing the environment, since it is only defined over the variables used for interacting with process p. This insight is the basis for deriving a transition function $E^{p,M,\phi}$ with variables only over those in T^p, yet $T^{\neq p} \implies E^{p,M,\phi}$.

To this end, let us first investigate how variables are shared in the component-oriented interpolation setting, as it is slightly different from the standard Craig interpolation setting in Figure 7.2. In the latter, there are three sets of variables. In the component-oriented setting, there are seven sets. Let us isolate environment transition step i, that is $T_i^{\neq p}(\bar{a}, \bar{b}, \bar{c}, \bar{d})$. The remainder environment transition steps are $T_{\neq i}^{\neq p}(\bar{e}, \bar{d}, \bar{c}, \bar{f})$ and the variables occurring in the component and property are $B(\bar{g}, \bar{b}, \bar{c}, \bar{f})$. This is showed as a Venn diagram in Figure 7.3.

We can now define a component-oriented interpolation scheme for $T_i^{\neq p}$:

Definition 7.2.1 (Component-Oriented Interpolant Construction from Resolution Refutation Graph)**.** Let Π be the resolution refutation for Equation (7.2) with the

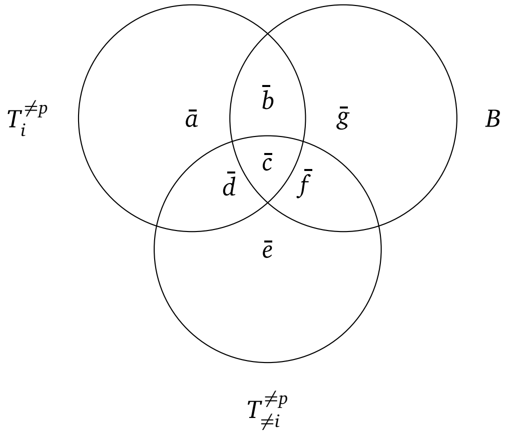


Figure 7.3: Relation of sets of variables occurring in the component-oriented setting.

graph $G_\Pi = (V_\Pi, E_\Pi)$ associated with it. Furthermore, let us partition the variables occurring in Equation (7.2) to the partitions shown in Figure 7.3. For each vertex $v \in V_\Pi$, let v_1 and v_2 be its predecessors and let C_v^i be a Boolean formula such that

$$\begin{array}{rcl}
\bot & \iff & v \in T_i^{\neq p} \text{ and } v \text{ is root} \\
\top & \iff & v \in T_{\neq i}^{\neq p} \text{ and } v \text{ is root} \\
\top & \iff & v \in I^{\neq p} \text{ and } v \text{ is root} \\
\top & \iff & v \in B \text{ and } v \text{ is root} \\
(\neg x \wedge C_{v_1}^i) \vee (x \wedge C_{v_2}^i) & \iff & x \in \bar{d} \cup \bar{c} \text{ and } x \text{ is the pivot variable of } v \\
 & & \text{and } x \in v_1 \text{ and } \neg x \in v_2 \text{ and } v \text{ is non-root} \\
C_{v_1}^i \vee C_{v_2}^i & \iff & x \in \bar{a} \cup \bar{b} \text{ and } x \text{ is the pivot variable of } v \\
 & & \text{and } v \text{ is non-root} \\
C_{v_1}^i \wedge C_{v_2}^i & \iff & x \in \bar{g} \cup \bar{f} \cup \bar{e} \text{ and } x \text{ is the pivot variable of } v \\
 & & \text{and } v \text{ is non-root}
\end{array}$$

If Definition 7.2.1 is applied from the leaf $\bot$, one gets a component-oriented interpolant for step i of the environment transition function, i.e. $T_i^{\neq p}$. The resulting is weak enough to preserve the over-approximation from Craig interpolation, i.e. $T_i^{\neq p} \implies C_\bot^i$, but not necessarily strong enough to strictly preserve the unsatisfiability of the formula from which the resolution refutation graph is derived, i.e. Equation (7.2). This is shown in the proof of the following theorem:

Theorem 7.2.2 (Over-Approximation by the Component-Oriented Interpolant). *Let σ be a valuation such that $\sigma(v) = \bot$ for any $v \in V$ in Definition 7.2.1. For any $i \in 1 \dots k$, the following holds:*

$$\sigma(C_v^i) = \bot \implies \sigma(a) = \bot \wedge a \in T_i^{\neq p} \tag{7.3}$$

furthermore, it also holds that

$$\sigma(C_v^i) = \top \implies \sigma(a) = \bot \wedge a \in T_{\neq i}^{\neq p} \cup I^{\neq p} \cup B \tag{7.4}$$

Proof. By induction similar in style to [Hel07], but modified for the component-oriented interpolation setting.

Consider the base case for Equation (7.3). If $v \in T_i^{\neq p}$, then $C_v^i = \bot$ by Definition 7.2.1 and therefore $\sigma(C_v^i) = \bot$. Also the hypothesis $\sigma(v) = \bot$ holds by definition. The same argument holds for proving the base case for Equation (7.4).

For the inductive step, we distinguish three cases:

1. Case where x is the pivot variable of vertex v and $x \in \bar{d} \cup \bar{c}$. Recall from Definition 7.2.1 that in this case, $C_v^i = (\neg x \wedge C_{v_1}^i) \vee (x \wedge C_{v_2}^i)$. Furthermore, recall from the definition of the resolution refutation graph in Section 7.1.1 that $v = D \vee D'$ and that $v_1 = x \vee D$ and $v_2 = \neg x \vee D'$. Given that $\sigma(v) = \bot$, it follows that $\sigma(D) = \sigma(D') = \bot$.

   Thus if $\sigma(C_v^i) = \bot$, there are two subcases, namely:

   - Case $\sigma(x) = \top$. Then $\sigma(C_{v_2}^i) = \bot$ and $\sigma(v_2) = \bot$. By induction we conclude that $\sigma(a) = \bot$ for some $a \in T_i^{\neq p}$.
   - Case $\sigma(x) = \bot$. Then $\sigma(C_{v_1}^i) = \bot$ and $\sigma(v_1) = \bot$. By induction we conclude that $\sigma(a) = \bot$ for some $a \in T_i^{\neq p}$.

   If $\sigma(C_v^i) = \top$, then there are the following two subcases:

   - Case $\sigma(x) = \top$. Then $\sigma(C_{v_2}^i) = \top$ and $\sigma(v_2) = \bot$. By induction we conclude that $\sigma(a) = \bot$ for some $a \in T_{\neq i}^{\neq p} \cup I^{\neq p} \cup B$.
   - Case $\sigma(x) = \bot$. Then $\sigma(C_{v_1}^i) = \top$ and $\sigma(v_1) = \bot$. By induction we conclude that $\sigma(a) = \bot$ for some $a \in T_{\neq i}^{\neq p} \cup I^{\neq p} \cup B$.

2. Case where x is the pivot variable of vertex v and $x \in \bar{a} \cup \bar{b}$. Observe that $C_v^i = C_{v_1}^i \vee C_{v_2}^i$ and that the premises for v, v_1, v_2, $\sigma(D)$ and $\sigma(D')$ hold likewise as for case 1.

   Given this, if $\sigma(C_v^i) = \bot$ then $\sigma(C_{v_1}^i) = \bot$ and $\sigma(C_{v_2}^i) = \bot$. Also here we can then distinguish two subcases:

   - Case $\sigma(x) = \top$ then $\sigma(v_2) = \bot$. By induction we conclude that $\sigma(a) = \bot$ for some $a \in T_i$.

- Case $\sigma(x) = \bot$ then $\sigma(v_1) = \bot$. By induction we conclude that $\sigma(a) = \bot$ for some $a \in T_i$.

It becomes more involved for the case that $\sigma(C_v^i) = \top$. We then know that either $\sigma(C_{v_1}^i) = \top$ or that $\sigma(C_{v_2}^i) = \top$ and that either $\sigma(v_1) = \bot$ or $\sigma(v_2) = \bot$. This leads to six possible subcases:

Subcase	$\sigma(v_1)$	$\sigma(v_2)$	$\sigma(C_{v_1}^i)$	$\sigma(C_{v_2}^i)$
(2.1)	$\top$	$\bot$	$\top$	$\top$
(2.2)	$\bot$	$\top$	$\top$	$\top$
(2.3)	$\top$	$\bot$	$\bot$	$\top$
(2.4)	$\bot$	$\top$	$\bot$	$\top$
(2.5)	$\top$	$\bot$	$\top$	$\bot$
(2.6)	$\bot$	$\top$	$\top$	$\bot$

Subcases (2.1), (2.2), (2.3), (2.6) trivially hold, as either $\sigma(v_1) = \bot$ and $\sigma(C_{v_1}^i) = \top$ or $\sigma(v_2) = \bot$ and $\sigma(C_{v_2}^i) = \top$ hold. In either case, we conclude that $\sigma(a) = \bot$ for some $a \in T_{\neq i}^{\neq p} \cup I^{\neq p} \cup B$.

For subcases (2.4) and (2.5), observe that the pivot variable x does not occur in $C_{v_1}^i$, $C_{v_2}^i$, $T_{\neq i}^{\neq p}$ nor in B. See Figure 7.3. This allows us to define a variation on the valuation σ which we denote as σ'. It holds the same valuations for any variable other than x, and for x it is defined as follows:

$$\sigma'(x) = \begin{cases} \bot & \text{if } \sigma(x) = \top \\ \top & \text{if } \sigma(x) = \bot \end{cases}$$

By using σ' instead of σ for case (2.4), we get $\sigma'(v_2) = \bot$ and $\sigma'(C_{v_2}^i) = \top$. By induction we conclude that $\sigma'(a) = \bot$ for some $a \in T_{\neq i}^{\neq p} \cup I^{\neq p} \cup B$. Note that $\sigma'(a) = \sigma(a)$ since x does not occur in $T_{\neq i}^{\neq p} \cup I^{\neq p} \cup B$. The same reasoning can be applied for case (2.5).

3. Case where x is the pivot variable of vertex v and $x \in \bar{g} \cup \bar{f} \cup \bar{e}$. This is the dual case to case 2 and is proven in a similar manner.

$\square$

Recall that C_v^i only derives a component-oriented interpolant for transition step i. By substitution of the occurring variables to current and successor-state variables, it could be already used as an abstraction for $T_i^{\neq p}$. It can be easily strengthend by accounting for the remaining steps in the bounded model checking formula. In general, the following corollary from Theorem 7.2.2 is used:

Corollary 7.2.3 (Component-Oriented Interpolated Environment Transition Condition)**.** *Let ϕ be the property of interest. Let $M = (I, T)$ be a composition of n processes such that $I = \bigwedge_{i=1}^{n} I^i$ and $T = \bigwedge_{i=1}^{n} T^i$. Let us assume that ϕ holds up to a given bound k, i.e. $M \models^k \phi$. Let process $p \in 1 \ldots n$ be the process of interest. The component-oriented interpolated transition condition, defined as $E^{p,M,\phi}$, can be derived from the interpolants $C_\perp^1, \ldots, C_\perp^k$ resulting from Theorem 7.2.2:*

$$E^{p,M,\phi} = \bigwedge_{i=1}^{k} C_\perp^i(\bar{r}_{i-1}, \bar{r}_i)[\bar{r}_{i-1}/\bar{s}, \bar{r}_i/\bar{s}']$$

It then follows that

$$T^{\neq p} \implies E^{p,M,\phi}$$

When additional abstractions of $T_i^{\neq p}$ are constructed using for example different techniques or by increasing k, these abstractions can be combined to obtain a stronger abstraction:

Proposition 7.2.4 (Conjuncting Abstractions of the Environment Transition Condition)**.** *Given the environment transition condition $T^{\neq p}$ and any two abstractions of it Y and Z such that $T^{\neq p} \implies Y$ and $T^{\neq p} \implies Z$, then the following holds:*

$$T^{\neq p} \implies (Y \wedge Z)$$

7.3 Applications

The component-oriented interpolation approach, as described in Section 7.2, enables several practical applications. We will describe three of them here. In the subsequent section, Section 7.4, we shall look at one of the applications, namely model verification, from an experimental perspective.

A straightforward application is manual inspection. With models as large and complex as the one in Chapter 6, manual inspection is more cumbersome to perform than employing model checking algorithms. Our component-oriented interpolation method can however overcome this. Assume that one is intimate with a particular component, or a set of components, but not with the remainder of the model. That remainder can be significantly abstracted by our component-oriented interpolation method to a remainder model that is expressed in the variables of the component one is familiar with. This makes such a remainder model smaller in size and thus more amenable for manual inspection. Abstractions can be combined by generating environments using various properties that (k-)boundedly hold on the overall model. Proposition 7.2.4 allows us to conjunct these remainder models together in a stronger environment.

The component-oriented interpolated transition condition can also be utilised for automated verification of properties. A naive approach would be to do bounded model checking up to a tractable depth k such that one obtains a resolution refutation graph. Then one picks a component p and use the component-oriented interpolation method to abstract the remainder. Heuristically, it is wise to include at least the components that are directly referred by the atomic propositions in the property of interest, since these directly affect the property of interest. The abstracted environment transition condition and the transition condition of p can then be subjected to unbounded model checking techniques. This abstracted model is smaller than the original model. The resulting abstracted model might however be too weak. If a counterexample is found during unbounded model checking, one has to distinguish whether it is a false-negative due to over-approximation, or whether it is a counterexample that also occurs in the original model. Techniques from CEGAR (counterexample guided abstraction refinement) [DKW08] can be applied to concretise the counterexample. Or the depth k can be increased, resulting in a larger and more informative resolution refutation graph from which a more precise component-oriented interpolant can be generated. Furthermore, CEGAR refinement techniques can be applied in conjunction as well.

We believe that the power of our component-oriented interpolation technique is most applicable during the construction of the model. In traditional model checking approaches, refinements or changes of the model need to be fully model checked again. Our technique can speed that up. Assume we want to refine or modify a component and leave, for the moment, its environment as it is. Also assume that the interface of the component does not change. We can use the resolution refutation graphs from the properties that hold on that model to construct a component-oriented interpolated transition condition of the environment. When the component that was intended to be refined or modified is changed, we can use our component-oriented interpolated transition condition for reverification of the properties instead of using the full blown model. The premise of using our component-oriented interpolated transition condition is that its faster, and can be reused provided the interface and the environment does not change. It supports a continuous (re)verification methodology during modelling and providing earlier feedback on the model under construction.

7.4 Experimental Evaluation

In this section we investigate the effectiveness of the abstractions of the component-oriented interpolation method in terms of abstraction convergence, computation time and memory consumption using two industrial-sized cases. The first one is exactly the same case from Chapter 6. The second case is a follow-up model of the same mission, but heavily refined with details from the critical design phase. The latter is thus more detailed, larger and by that extend more complex. The report on the latter is at the time of writing yet unpublished. The experimental setup takes on verification as the application for our method. The data obtained from this can be used to infer the potential for other applications, like manual inspection and continuous reverification.

7.4.1 Case Configurations

In our evaluation, we considered several configurations from the preliminary design review and the critical design review satellite architecture model. From here on, we shall refer to the former as the PDR model and the latter as the CDR model. The final configurations selected for our experimental evaluation are known to require an interesting k for proving or disproving the property, as there were also configurations whose property were trivial to check. These final case study configurations are outlined in Table 7.1.

Model	Fault Injections	Property
PDR-1	Earth sensor failure	fail-operational flag is set
PDR-2	Propulsion failure	AOCS status flags are consistent
CDR-3	Various platform failures	not in safe mode
CDR-4	(none, i.e. nominal behaviour)	solar voltage level is consistent
CDR-5	(none, i.e. nominal behaviour)	not in safe mode

Table 7.1: Overview of experiment configurations in terms of the used model, the applied fault injections and the verification property. The first two configurations are from the PDR model, whereas the remainder three are from the CDR model.

7.4.2 Implementation

We measured the computation times and peak memory consumption of our technique against two other techniques. They were implemented in NuSMV 2.5.4 using MiniSAT 1.14p with a proof-logging extension as the SAT-solver.

We intended to use NuSMV's BDD-based verification implementation as the baseline. We however quickly found out that the BDDs were not effective on both the PDR and CDR configurations. On the PDR configurations, its performance was a magnitude (order) slower than the other techniques. On CDR configurations, the time needed to build the BDD of the transition function took more time than the overall computation time of the other techniques. We therefore omit the results from it.

Instead, we use McMillan's interpolation-based unbounded model checking technique for invariants [McM05] as the baseline. Its base principle is to interpolate the first transition step $A = R \wedge T(x_0, x_1)$ with $B = T(x_1, x_2) \wedge \ldots T(x_{k-1}, x_k) \wedge \neg\phi$ where ϕ is the invariant and initially $R = I_0$. The interpolant C, with $A \implies C$, is defined over variables in x_1 and is thus a weakened characterization of the successor states. This process is repeated with taking $R \leftarrow R \vee C$ as the initial states until the full state space has been explored. A sketch of the algorithm is shown in Algorithm 1. We implemented it in NuSMV as there was not pre-existing implementation for it available. The interpolation scheme we implemented is by McMillan as well [McM03] and it has been studied thoroughly for use in this setting [DSi+10; RSS12].

Algorithm 1 McMillan's Interpolation-based Invariant Checking.

```
k ← 1
while $I_0 \wedge T_1 \wedge \cdots \wedge T_k \wedge \neg\phi$ is unsatisfiable do
    $R \leftarrow I_0$;
    while $R \wedge T_1 \wedge \cdots \wedge T_k \wedge \neg\phi$ is unsatisfiable do
        $C \leftarrow$ interpolant of $R \wedge T_1$
        if $C \wedge \neg R$ is satisfiable then $R \leftarrow R \vee C$
        else[no new states explored] return $\phi$ holds
        end if
    end while
    $k \leftarrow k + 1$
end while
return counterexample
```

The component-oriented interpolation technique was put into a verification scheme. The algorithm, shown in Algorithm 2 bears similarity with McMillan's Interpolation-Based Invariant Checking scheme in Algorithm 1. Intuitively, it obtains a component-oriented interpolated environment, which is then used in an inner reachability analysis until a fixpoint is found, in which case the property holds. Otherwise, the bound is increased in the hope for a stronger component-oriented interpolated environment. Even though any reachability algorithm could be used, we employed McMillan's interpolant-based invariant checking algorithm here. The primary reason for this design decision is that we stay in a SAT-based context and the data-structures associated with it. If any other context would be used, like for example BDDs, we would have to convert $E^{p,M,\phi}$, I^p and T^p to the data-structures associated with them, resulting in additional overhead.

7.4.3 Experimental Data

The experimental data was obtained from a machine with a 2.33 GHz CPU and 32 GB RAM. A summary of the data is presented in Table 7.2. We kept track of the depth needed to determine whether the property holds or whether there exists a counterexample. This bound is the column k in Table 7.2. A smaller k would indicate a faster convergence of the abstraction.

The results indicate that the complexity of the CDR model is higher than the PDR model. This is not surprising since the CDR model is a refinement of the PDR model by having more behavioural detail. The results furthermore indicate that the verification by the component-oriented interpolation method is competitive.

Algorithm 2 Component-Oriented Interpolation-based Invariant Checking.

```
k ← 1
while $\neg\phi \wedge I_0^p \wedge T_1^p \wedge \cdots \wedge T_k^p \wedge I_0^{\neq p} \wedge T_1^{\neq p} \wedge \cdots \wedge T_k^{\neq p}$ is unsatisfiable do
    $E^{p,M,\phi}$ ← component-oriented interpolant of $I_0^{\neq p} \wedge T_1^{\neq p} \wedge \cdots \wedge T_k^{\neq p}$
    $R \leftarrow I^p$
    while $R \wedge T_1^p \wedge E_1^{p,M,\phi} \wedge \cdots \wedge T_k^p \wedge E_k^{p,M,\phi} \neg\phi$ is unsatisfiable do
        $C$ ← interpolant of $R \wedge T_1^p \wedge E_1^{p,M,\phi}$
        if $C \wedge \neg R$ is satisfiable then $R \leftarrow R \vee C$
        else[no new states explored] return $\phi$ holds
        end if
    end while
    $k \leftarrow k + 1$
end while
return counterexample
```

Case	Technique	Outcome	k	Time (sec)	Mem (Mb)
PDR-1	MCM	counterexample	3	2.42	95.9
	COMP	counterexample	3	3.52	111.9
PDR-2	MCM	counterexample	2	1.77	92.0
	COMP	counterexample	2	2.28	100.4
CDR-3	MCM	counterexample	11	486.06	651.0
	COMP	counterexample	11	338.56	865.5
CDR-4	MCM	holds	4	7.10	125.7
	COMP	holds	3	7.00	138.0
CDR-5	MCM	holds	7	69.20	171.5
	COMP	holds	3	8.10	137.0

Table 7.2: Summary of verification outcome, needed depth k, verification time and peak memory consumption for McMillan's interpolation-based invariant checking (MCM) and the component-oriented interpolation-based invariant checking (COMP).

This is in particular visible for CDR-3 and CDR-5, where the computation time is better. The needed depth k indicates that the quality of the abstraction is also competitive. This is visible when the property holds, namely cases CDR-4 and CDR-5, where a smaller k is required. Note that these measures cannot be trivially generalised. The timings depend heavily on the used SAT-solver and in particular the heuristics it employs. A change in the clause orderings could make a difference, or simply running the same case on a different system could lead to differences. These factors are inherent to the nature of current-day SAT-solvers. Hence, the numbers should be interpreted as indications, rather than hard conclusions.

Even though the experimental data indicate that the quality of the abstraction by the component-oriented interpolation scheme is competitive, we suspect that the way it is used in this experimental evaluation suffers from double abstraction. This comes from the fact that $E^{p,M,\phi}$ is an abstraction, which, according to Algorithm 2, is used for an inner unboundend reachability check. The latter also uses abstraction, namely by abstracting reached successor states. This might cause more false-negative counterexamples than necessary. Each abstract counterexample turns the while condition in line 5 of Algorithm 2 to false, leading to an increase of the bound k in line 9. An exact and unbounded inner reachability check would be preferable. We are however not aware of any that employ SAT-based setting. BDD representations on the other hand are possible. We experimented with a quick implementation, but early tests indicated that the benefit of the exactness of BDDs were quickly diminished by the time needed to construct the initial condition, the component and its environments as BDDs. Hence, we leave any further optimisation in this area as future work.

As elaborated in Section 7.3, there are other uses, like manual inspection or reverification, where the algorithm looks slightly different than that for plain verification, as described in Algorithm 2. For manual inspection, it is important that bounded verification (line 2 of Algorithm 2) and the construction of the component-oriented interpolated environment condition (line 3 of Algorithm 2) are fast. For reverification on the other hand, it it is more important that the inner unbounded verification is performant (lines 5 to 8). To estimate these factors, we kept track of the timed used on bounded verification, component-oriented interpolation and inner unbounded verification. The result is shown in Table 7.3.

The table shows that component-oriented interpolation takes relatively little time. This is expected. The worst-case time-complexity of constructing is linear to the size of the resolution refutation graph and the bound k. Bounded verification and inner unbounded verification, in our experiment, are based on SAT-solving, which is known to be NP-complete.

In case the property holds, cases CDR-4 and CDR-5, the inner unbounded reverification takes a large chunk of the computation time. The gain is however when reverification is performed often, as the time for bounded verification and component-oriented interpolation is avoided. The benefit would be even greater if the needed depth is larger, as one then avoids the costly bounded verification.

Case	Depth	BV (msec)	COI (msec)	IUV (msec)
PDR-1	1	112	36	256
	2	284	76	512
	3	588	-	-
PDR-2	1	152	36	272
	2	436	-	-
CDR-3	1	264	68	412
	2	568	152	960
	3	1108	236	1596
	4	1876	332	2612
	5	3633	468	5424
	6	5116	620	3905
	7	12592	989	5048
	8	14309	1348	6544
	9	66244	5597	16237
	10	94174	6900	29014
	11	38127	-	-
CDR-4	1	168	52	340
	2	360	96	652
	3	564	152	2744
CDR-5	1	168	40	260
	2	364	92	632
	3	588	148	3964

Table 7.3: Decomposition of the verification times of Table 7.2 into the three most time consuming parts per depth, namely bounded verification (BV), component-oriented interpolation (COI) and inner unbounded verification (IUV).

7.5 Discussion

The verification of properties has been an important driver to the exploitation of a model's compositionality for tackling the state space explosion. Approaches have been devised that exploit component interactions through global variables, whereas others have tackled the notion of shared variable interactions. Also expressiveness regarding invariants, safety and liveness properties have been studied. In this section, we shall discuss the closely related works.

Closely related, and the work that inspired us initially, is that by [CN09]. Their approach is based on the computation of the split invariant, which is a conjunction of process invariants. The safety property of interest is then checked against the split invariant. It is an over-approximation, as it accounts for interference by other processes using a Cartesian product of possible local states. Due to the over-approximation, counterexamples could be false negatives. Refinement occurs by analysing the counterexample and by adding auxiliary global variables that result to a finer over-approximation. The approach works well for systems that have a low ratio of global-local variables. Global variables are those shared with *all* processes occurring in the model. In our case, that would result to having all data ports becoming global variables, even though they are not visible to all components. This would counterfeit the benefits of the approach, making it force to see it as a global model checking problem. The same authors extended this work to liveness properties in [CN08]. The constraint is that the atomic propositions in the liveness property is only allowed over global variables. The split invariant computation is modified to synthesise environment processes for each processes. Our notion of environment (cf. Definition 7.2.1) was inspired by this approach. The environment process accounts for the possible interference by other threads. As processes can only interfere through global variables, the environment process operates only over those. This in contrast to our approach, where processes interfere through shared variables. In both their and our approach, the environment process is an over-approximation. We attempt to refine it by increasing the bound k, whereas the approach by [CN08] adds auxiliary variables. In [Coh+11], it is reported that the method works well on small benchmarks, and they also show that their approaches are effectively paralellisable over multi-core systems.

Another branch of compositional verification that bears resemblance to the works of [CN09] is called "thread-modular" and initiated from [FQ03]. In this work, it is identified that for *loosely-coupled* concurrent processes with global/-local variables, environment assumptions can be inferred that account for the interference by other threads. These assumptions are similar to the environment processes of [CN08]. Their method is incomplete as well, since the environment assumptions over-approximate, and no refinement procedure is described. Later on, in [MPR06], those authors show that Cartesian abstract interpretation and thread-modular verification are conceptually the same, but were developed in different communities and differ in the details. In a more recent work [GPR11b], it is shown that the Owicki-Gries paradigm, that is the underpinning of [CN09], is also conceptually the same as thread-modular verification, and by extension also to Cartesian abstract interpretation. They are share the principle that a state space per process is explored, and hence our work can also be classified to this branch. The difference is how they account for interference by other processes. In the Owicki-Gries approaches by [CN08; CN09], process interference is an over-approximation and is refined through counterexample analysis and the addition of auxiliary variables that coarsen the interference. For the thread-modular approach, an refinement approach using so-called exception sets is described in [MPR10]. In [GPR11a], a refinement approach is described by setting up a set of Horn clauses, solve it, and extract refined transition predicates from it. All these lines of work only consider the notion of global variables for communication and cannot cope with a finer notion of process interaction, like through shared variables. Our approach fits both.

Another line of research that takes a compositional perspective is that of assume-guarantee reasoning, or also known as rely-guarantee reasoning. They started out as a technique for proving properties manually, but have been automatised in recent years. There is an enormous body of work in this line. We will restrain ourselves to the most related approaches. Akin to [CN08], the principle idea is to generate an environment for each process, which describes assumed behaviour from its environment (that is, the remainder processes). In return, the process provides guarantees, which can function as an assumption for other processes. The simplest version of assume-guarantee reasoning is the asymmetric AG-rule. According to this rule, a weakest assumption is generated based on the property and one of the components. This assumption is then used as the guarantee for the remainder components. In [GPB02], a technique based on automata determinisation is described to generate weakest assumptions. In subsequent work [CGP03], assumptions are learned using an automaton learning algorithm, like L*. In many cases, components mutually influence each other, rendering a proper application of the asymmetric AG-rule infeasible. A solution was proposed, called symmetric AG-rules [BGG03]. It is a slight adaptation of the asymmetric rule that accounts for the idea that all components can have assumptions, provided that the composition of assumptions is part of the original property. The work by [Cof+12] follows this approach for component architectures, including AADL (and alike). In literature, typically models with global/local variables were tackled. An exception to this is the work described in [Lom+10]. It assumes that the global property to be verified is a conjunction of local properties. The work provides a bounded assume-guarantee reasoning rule that accounts for effects by transitivity of the component's topology. The drawback of this work is the assumption: many requirements are simply not the conjunction of of local requirements. The aforementioned approaches are suitable for breaking up the state space, and reason over it compositionally, avoiding the state space explosion memory-wise. It however does not have the benefit of reasoning over parts of the state space in parallel, since the AG-rules have to applied in a particular component order.

The works discussed so far relate themselves to our work by tackling the problem of composition/decomposition of verification. The use of interpolation to do this is novel. Interpolation itself however has been used before to non-compositional model checking. In [McM05], Craig's interpolation theorem is used to abstract the reachable states with respect to a safety property. It does this by abstracting the one-step reachable states, such that it is a sound over-approximation with respect to the property. It is however not an over-approximation in general. The one-step reachable states over-approximation is computed based on a k-bounded bounded model checking formula. It is complete, and known to terminate as long as a sufficient k is provided. An interpolation based technique for model checking LTL is described in [McM03]. It encodes the automata-based approach towards LTL model checking and uses interpolation to over-approximate the sets of currently explored states. In [JM05], Craig's interpolation theorem is applied in a different way, namely for abstracting the transition relation itself. They showed it works well in conjunction with predicate abstraction. We built upon these works by applying interpolation to exploit a composition of communicating processes.

8

Krylov-Based Transient Analysis of Continuous Time Markov Chains

The predominant technique for computing the transient distribution of a Continuous Time Markov Chain (CTMC) exploits uniformisation, which is known to be stable and efficient for non-stiff to mildly-stiff CTMCs. On stiff CTMCs however, uniformisation suffers from severe performance degradation. In this chapter we reintroduce a Krylov-based method for computing the transient of a CTMC. It is briefly mentioned in Moler and Van Loan's discourse [ML03] on 19 methods for the matrix exponential as a novel 20th method and in De Souza e Silva and Gail's survey [dSeSG99] as a possible method for computing the transient of CTMC. Despite these references and their success for many matrix-related computations in different fields of science and engineering, Krylov-based methods received scant attention in the field of probabilistic analysis. We believe this is due to three reasons, namely (i) to our knowledge, experiments with a Krylov-based method have been only conducted on small academic examples [SSS96] or without regard to stiffness versus non-stiffness (ii) due to the lack of the former, nobody has identified the class of CTMCs for which Krylov-based methods excel and (iii) the good applicability of Krylov-based methods to the transient have, to our knowledge, not been explained theoretically. In this chapter, we tackle these reasons. First, we show how to apply a Krylov-based method for computing the transient distribution of CTMCs to model check time-bounded reachability properties expressed in Continuous Stochastic Logic (CSL). Then using five case studies from literature, we extensively compared the implemented Krylov-based method to the existing uniformisation-based method. From the results, we identified that computing the transient distribution is (much) faster with Krylov-subspace methods for stiff CTMCs. This observation is explained by the good approximation properties of the Krylov-based matrix exponential using Schwerdtfeger's formula [Rin55].

8.1 Stiffness

Stiff CTMCs are found in many domains, among which systems biology, where the reaction rates of molecules may vary greatly, and mission critical systems engineering, where failures occur frequently (like sensor glitches) or sporadically (like complete sensor failure). The transient distribution of CTMCs —what is the probability to be in a state at time t?— is a prominent measure of interest, and is fundamental to a range of measures of interest such as time-bounded reachability properties [Bai+03]. Its computation is a well-studied topic and a survey of applicable techniques is discussed by De Souza e Silva and Gail [dSeSG99]. One wide-spread method is Jensen's uniformisation [Jen53] which is known for its good numerical stability and is implemented as the default method for transient analysis in various —if not all— Markov analysis tools. Its performance degrades however on stiff models, which, given its many definitions in literature, we simply refer to as the degree of difference between the smallest and largest rates in the CTMC. Other methods like Runge-Kutta solvers require small discretisation values on stiff models, thereby suffering from similar performance problems. On top of these problems, potential numerical instability, not uncommon with stiff models, needs to be dealt with as well.

8.2 Model Checking Markov Chains

This section introduces the basic concepts of model checking CTMCs using Continuous Stochastic Logic (CSL). It is only used as a stepping stone towards the remainder part of this chapter. We refer to [Bai+03] for an elaborate treatment on this topic.

A labelled CTMC is a tuple $(S, \mathbf{Q}, L)$ where S is a finite set of states, $L : S \rightarrow 2^{AP}$ is a labelling function and $\mathbf{Q} : S \times S \rightarrow \mathbb{R}$ is a generator matrix. Each diagonal element $q_{s,s} \in \mathbf{Q}$ is defined as $q_{s,s} = -\sum_{s' \in S, s \neq s'} q_{s,s'}$, and all remaining elements $q_{s,s'}$ have a rate ≥ 0. Intuitively, a transition from s to s' (with $s \neq s'$) is triggered within t time units by probability $1 - e^{-q_{s,s'} t}$. In other words, the occurrence of a transition is exponentially distributed. The rate of staying in a state s is described by the diagonal elements, namely $|q_{s,s}|$.

The *transient* distribution, which is further referred to in this dissertation as the transient, of a CTMC, denoted by $\pi(t)$, is the vector of probabilities being in states $s \in S$ at a time t given an initial distribution $\pi(0)$. It is characterized by Kolmogorov's forward differential equation $\frac{d}{dt}\pi(t) = \mathbf{Q} \cdot \pi(t)$, whose solution, given an initial distribution $\pi(0)$, is the following:

$$\pi(t) = e^{\mathbf{Q}t} \cdot \pi(0) \tag{8.1}$$

There are numerous numerical techniques to compute $\pi(t)$, of which Jensen's uniformisation algorithm [Jen53] is widely used.

Uniformisation considers a uniformisation rate $\Lambda \geq max_{i \in S}|q_{i,i}|$ so that the generator matrix can be rewritten as $\mathbf{Q} = \Lambda \cdot (\mathbf{P} - I)$. The matrix $\mathbf{P}$ is a stochastic matrix of the *uniformised CTMC*, and I is the identity matrix. When this rewritten $\mathbf{Q}$ is substituted in Equation (8.1), we get $\pi(t) = e^{\Lambda(\mathbf{P}-I)t} \cdot \pi(0)$. This equation can be rewritten and the matrix exponential can be expanded according to the Taylor-MacLaurin series, after which one gets:

$$\pi(t) = (\sum_{n=0}^{\infty} e^{-\Lambda t} \frac{(\Lambda t)^n}{n!} \mathbf{P}^n) \cdot \pi(0) \tag{8.2}$$

The part $\sum_{n=0}^{\infty} e^{-\Lambda t} \frac{(\Lambda t)^n}{n!}$ is the Poisson density function and it converges to 1. A numerically stable technique for computing it is by Fox-Glynn's method [FG88]. When an error bound $\epsilon > 0$ is given, the sum of Equation (8.2) can be truncated. The error bound ϵ can be used to determine the left- and right series truncation points $\mathscr{L}_\epsilon$ and $\mathscr{R}_\epsilon$, such that $\sum_{\mathscr{L}_\epsilon}^{\mathscr{R}_\epsilon} e^{-\Lambda t} \frac{(\Lambda t)^n}{n!} \geq 1 - \epsilon$. The left and right truncation points tend to be in the order of $O(\Lambda t)$. Large Λ's are common for stiff CTMCs and if this is also combined with a large t, the number of terms needed by uniformisation to compute the transient is large.

The transient is fundamental to analyse labelled CTMCs with properties expressed in CSL, which describes a measure of interest in terms of satisfiable states and paths. It is also at the heart of more recent verification techniques that check a CTMC against a timed automaton specification [Che+09]. For the scope of this chapter, the interesting CSL properties are of the form $\mathscr{P}_{\bowtie p}(\diamondsuit^{[t_1,t_2]}\Psi)$. Intuitively, it means that the set of paths that eventually reach a state satisfying Ψ has a probability measure meeting $\bowtie p$ (where $\bowtie \in \{<, >, \leq, \geq, =\}$) within the real-valued time bounds t_1 to t_2. Ψ is a CSL formula (in all our examples a boolean expression) over the set of atomic propositions *AP* used in the labelled CTMC. To evaluate these kind of CSL properties, one computes the transient on a modified labelled CTMC(s) and compares the transient probabilities with the bound $\bowtie p$.

8.3 Krylov Subspace Methods

In the remainder of this chapter, we use $A = \mathbf{Q} \cdot t$ and $v = \pi(0)$, to keep the notation similar to the literature of Krylov-subspace methods [Saa92] while maintaining a connection to the matrix exponential in Equation (8.1).

The principal idea of Krylov subspace methods is to approximate the original sparse matrix A by a matrix H_m of much smaller dimension m. This works because H_m preserves an important property of A: its extreme eigenvalues. We will show using Schwerdtfeger's formula [HJ86] that due to the extreme eigenvalue preservation, $e^A v$ can be effectively approximated by operations on H_m.

8.3.1 Mathematical Formulation

A naive approach for computing $e^A v$ is by using the Taylor-MacLaurin series expansion:

$$e^A v = \sum_{i=0}^{\infty} \frac{A^i}{i!} v = Iv + Av + \frac{1}{2} A^2 v + \ldots$$

The matrix powers make it evident that this approach is highly numerically unstable. Fortunately, numerous stable techniques have been developed by the numerical linear algebra community. A powerful technique central in this chapter, Krylov-based methods, exploits the sparseness of the matrix. This property typically holds for infinitesimal generators. Several researchers [HL97; Saa92] have developed and studied Krylov-based methods to the matrix exponential, where the principal idea is to approximate $e^A v$ by an element in the m-order Krylov subspace, defined as

$$K_m(A, v) = \text{span}\{v, Av, A^2 v, \ldots, A^{m-1} v\}$$

where span denotes the usual linear span of a set of vectors. The precision of the approximation is controlled by the natural m. A lower m leads to a coarser approximation while a higher m increases precision at the expense of increased memory and computation time.

Algorithm 3 Arnoldi iteration.

```
$v_1 \leftarrow v/||v||_2$
for $j = 1, 2, \ldots, m$ do
    $w \leftarrow Av_j$
    for $i = 1, 2, \ldots, j$ do
        $h_{i,j} \leftarrow (w, v_i)$
        $w \leftarrow w - h_{i,j} v_i$
    end for
    $h_{j+1,j} \leftarrow ||w||_2$
    $v_{j+1} \leftarrow w/h_{j+1,j}$
end for
```

The approximation to $e^A v$ starts with the Arnoldi iteration, which is shown in Algorithm 3. In this figure, the dot product of two vectors w and v_i is denoted as (w, v_i) and the Euclidean norm of a vector w is denoted as $||w||_2$. The iteration produces a sequence of orthonormal Arnoldi vectors v_1 through v_m, which as a matrix V_m forms the orthonormal basis of the Krylov subspace K_m. It also produces the matrix H_m from the coefficients $h_{i,j}$. That matrix is the linear projection of A onto subspace K_m and is of upper Hessenberg form, i.e. H_m is nearly triangular due to the non-zero entries in first subdiagonal. Thus what the Arnoldi iteration does is a Hessenberg decomposition of A, resulting in the following relation:

$$A \approx V_m H_m V_m^T$$

From this decomposition, the computation of $e^A v$ can be derived by operations on the smaller H_m. In this derivation, we use e_n, which is the n^{th} vector of I:

$$\begin{aligned}
e^A &\approx e^{V_m H_m V_m^T} && \textit{(by application of exponential)} \\
e^A &\approx I + V_m H_m V_m^T + \frac{1}{2}(V_m H_m V_m^T)^2 + \ldots && \textit{(by series expansion)} \\
e^A &\approx V_m(I + H_m + \frac{1}{2}H_m^2 + \ldots)V_m^T && \textit{(by } I = V_m^T V_m = V_m V_m^T\textit{)} \\
e^A &\approx V_m e^{H_m} V_m^T && \textit{(by series de-expansion)} \\
e^A V_m &\approx V_m e^{H_m} && \textit{(by multiplication with } V_m\textit{)} \\
e^A v_1 &\approx V_m e^{H_m} e_1 && \textit{(by } v_1 = V_m e_1\textit{)} \\
e^A v &\approx V_m e^{H_m} e_1 ||v||_2 && \textit{(by } v_1 = v/||v||_2\textit{)}
\end{aligned}$$

The last equation means that one can approximate the exponential over matrix A by computing the exponential over the much smaller H_m using stable dense methods (like Padé approximation) and project the result back to the original space using matrix V_m.

There are several advantages to this approach. First, the method can be performed iteratively. If the precision does not suffice for a particular subspace dimension m, this can be increased and the Arnoldi iteration can resume with the existing matrices V_m and H_m and iteratively extend them until a satisfactory precision has been reached. The second advantage is the numerical robustness. During the Arnoldi iteration, only multiplication, addition, division and subtraction is performed on normalized vectors. The exponential over H_m is stable when Padé approximation is combined with scaling and squaring, as was established for example by Ward [War77].

8.3.2 Complexity Analysis

The time-complexity of the Krylov-based method is dominated by the complexity of Arnoldi iteration. Arnoldi iteration has a time-complexity of $O(m \cdot n^2 + m^2 \cdot n)$, where m is the Krylov subspace dimension and n is the dimension of the matrix A. Line 1 of Algorithm 3 takes $O(n)$. Line 3 of Algorithm 3 is a matrix-vector multiplication, which is has a quadratic time-complexity. The for-loop between lines 4 to 6 is traversed at most m times, where line 5 and 6 are $O(n)$. Thus the inner-for loop takes $O(m \cdot n)$. Lines 8 and 9 take $O(n)$. Putting all these together, the result is a complexity of $n + m \cdot (n^2 + m \cdot n + 2 \cdot n)$, whose asymptotic time complexity is $O(m \cdot n^2 + m^2 \cdot n)$.

Note that Arnoldi iteration only computes the decomposition $A \approx V_m H_m V_m^T$. For computing $V_m e^{H_m} e_1 ||v||_2$, a matrix exponential needs to be computed over H_m. In our experiments, we used Padé approximation combined with scaling and squaring. It costs $2 \cdot (p + \gamma + \frac{1}{3}) \cdot m^3$ operations, where p is the degree of the Padé approximation, γ is the squaring & scaling factor [War77]. A larger p gives better precision, and typically a value > 6 is recommended [Sid98]. We used 16 in our experiments (cf. Section 8.4). The value γ determines the degree of scaling & squaring which is used to control numerical stability. It is suggested by [War77] to choose a γ such that $2^{\gamma-1} \leq ||H_m||_1 \leq 2^{\gamma}$, where $||H_m||_1$ is the 1-norm (maximum absolute column sum) of H_m. The resulting time complexity for Padé approximation is $O(\gamma \cdot m^3)$.

8.3.3 Schwerdtfeger's Formula

The approximation of $e^A v$ by $V_m e^{H_m} e_1 ||v||_2$ works particularly well, despite H_m being of much lower dimension than A. Attempts to explain this behaviour have led to advances in determining stricter error bounds [HL97; Saa92]. Instead of taking that direction, we shall explain it by an analysis in terms of eigenvalues of A and H_m.

A great deal of study has been conducted in relation of the eigenvalues of H_m to those of A. It is now well-accepted that H_m's eigenvalues, referred to as Ritz values, strongly correspond to the extreme eigenvalues of A [TBI97]. Those are the eigenvalues near the edge of A's spectrum. We will show that those are the eigenvalues of interest for the matrix exponential.

Any analytical function over a matrix A, like the exponential, can also be described in terms of the eigenvalues of A. Several theorems for this exist and Rinehart has shown that they are derivable to each other [Rin55]. Here we choose Schwerdtfeger's formula because its notation fits well in this context. When it is applied to the exponential, the following formula holds

$$e^A = \sum_{j=1}^{t} A_j \sum_{k=0}^{s_j-1} \frac{e^{\mu_j}}{k!} (A - \mu_j I)^k \tag{8.3}$$

where $\mu_1, \ldots, \mu_t$ are the distinct eigenvalues of A and $s_1, \ldots, s_t$ are the corresponding multiplicities. The term A_j is the Frobenius covariant [HJ86] associated with eigenvalue μ_j. It is computed using the corresponding left eigenvectors $x_1, \ldots, x_{s_i}$ and right eigenvectors $y_1, \ldots, y_{s_i}$ via summation: $A_j = \sum_{k=1}^{s_i} x_k y_k$.

The term e^{μ_j} in Equation (8.3) exponentially converges to zero for small μ_j. This novel insight explains the good approximation of the Krylov-based matrix exponential: *only the largest eigenvalues, preserved by H_m, are dominant for the matrix exponential.* This observation coincides with a result by Garren and Smith [GS00], who concluded that the second largest eigenvalue (the largest eigenvalue is always one) is a good estimator for the convergence to the steady state. Equation

(8.3) backs this result, indicating that the second largest eigenvalue is the most dominant for the transient behaviour, thus also for the steady state.

8.3.4 Error Estimates

Krylov-based methods are approximations and those come with a certain loss of information. The study of the error induced by Krylov-based methods is an extensively fast-moving field. Yet the current a-priori error bounds are known to be overly conservative [HL97; Saa92] for linear applications of Krylov-based methods, let alone for Krylov-based matrix exponentials. For this reason, Saad studied a-posteriori error estimates [Saa92]. They are based on truncation of the real error $e^{A}v - V_m e^{H_m} e_1 ||v||_2$, which is the following:

$$h_{m+1,m} \sum_{k=1}^{\infty} e_m^T \phi_k(H_m) e_1 A^{k-1} v_{m+1} \tag{8.4}$$

The function ϕ_i is defined by the recurrence relation

$$\begin{aligned} \phi_0(z) &= e^z \\ \phi_{i+1}(0) &= 1 \\ \phi_{i+1}(z) &= \frac{\phi_i(z) - \phi_i(0)}{z} \end{aligned}$$

Note that $\phi_{i+1}(0) = 1$ is defined by continuity, making the function ϕ well defined and analytic for all z. Based on the series of Equation (8.4), Saad proposes several error *estimates* because sharp error bounds are too conservative. All estimates are under-approximations of the real error because they are based on norms of the series's first terms.

An exception to this is Saad's second estimate, which is described as a *rough* estimate. It is defined as the first term of the series in Equation (8.4) with $\phi_1(H_m)$ replaced by e^{H_m}, because the latter is cheaper to compute (and already computed). The resulting error estimate is the following:

$$h_{m+1,m} \left| e_m^T e^{H_m} ||v||_2 e_1 v_{m+1} \right| \tag{8.5}$$

Saad provides little argumentation why it is safe to approximate $\phi_1(H_m)$ by e^{H_m}. The latter is actually always bigger than the former, which we shall prove as follows.

The first recursion of $\phi_i(z)$ can be rewritten as the following series:

$$\begin{aligned}\phi_1(z) &= \frac{e^z-1}{z}\\ &= \frac{1}{z}\sum_{k=1}^{\infty}\frac{z^k}{k!}\\ &= \sum_{k=1}^{\infty}\frac{z^{k-1}}{k!}\\ &= \sum_{k=0}^{\infty}\frac{z^k}{(k+1)!}\end{aligned}$$

It is not difficult to see that the right-most equation is always smaller than the Taylor-MacLaurin series of e^z. The experimental data from Saad's study suggest that Equation (8.5) is a bounded over-approximation of the real error, but this result is left unproven. Nevertheless, the study shows it is empirically a good estimate and for this reason it was used as the error estimate in our experimental evaluation.

8.4 Experiments

To compare the Krylov-based computation of the CTMC transient distribution against the uniformisation-based method, we implemented the former in the Markov Reward Model Checker (MRMC) [Kat+11]. Uniformisation is set as the default numerical engine of MRMC. We made a selection of case studies from the literature describing models from system biology, queuing networks and communication protocols and ran MRMC for different configurations of each case study for comparison.

8.4.1 Implementation

The Krylov-based method was implemented as an extension to MRMC by intercepting the invocations to uniformisation. It reuses the already implemented Harwell-Boeing sparse matrix data structure [DGL92] to store the infinitesimal generator matrix. The Krylov project matrix V_m and the Hessenberg matrix H_m are dense and were stored using the existing matrix data structures from the GNU Scientific Library (GNU GSL).

As there is no effective method (yet) to decide the perfect subspace size m given a particular error ϵ, the Krylov-based method was implemented as an iterative algorithm by repeatedly incrementing m until the desired error level is reached (see Section 8.3).

8.4.2 Experimental Setup

All experiments were run on a cluster of twelve identical nodes. Each node is equipped with a 2.33 GHz processor and 16GB RAM. The loaded operating system is 64-bits OpenSuSE 10.3. The cluster is only used for distributing the isolated runs over the nodes to speed up the overall experiment. For all case studies, three different configurations were run and an error level of 10^{-6} was used:

UNI	These are runs with MRMC's default numerical engine, uniformisation, enabled and steady-state detection [Kat+11] disabled.
UNI-S	These runs are similar to the previous, but with steady-state detection enabled.
KRY	These are runs with the iterative Krylov-based transient implementation as described in the previous section.

8.4.3 Case Studies

A careful selection of case studies from literature was made to comprise different modelling domains, different model sizes and different degrees of stiffness.

CSPS	A cylic server polling system that consists of $N = 5$ stations. The model was originally described by Ibe and Trivedi [IT90]. The measure of interest is the probability that given an upper timebound, the second station will eventually start serving. This expressed in CSL as $\mathscr{P}_{=?}(\Diamond^{[0,t]}\mathit{full})$.
TQN	A tandem queueing network with capacity $c = 20$ described by Hermans et al. [HMS99]. The measure of interest is the probability that the first queue sc will become full within t time units. This is expressed in CSL as $\mathscr{P}_{=?}(\Diamond^{[0,t]}sc = 20)$.
PTP	A simple peer-to-peer file sharing protocol described by Kwiatkowska et al. [KNP06]. The swarm consists of one client that already has all $K = 5$ blocks of the file and $N = 2$ other clients that have obtained no blocks so far. The measure of interest is whether all N clients have obtained all K blocks by time t. This is expressed in CSL as $\mathscr{P}_{=?}(\Diamond^{[0,t]}\mathit{done})$.
ER	An enzymatic reaction model by Busch et al. [BSW06]. It describes the enzyme-catalysed conversion of a molecular substrate species. The measure of interest is the probability that four units of the product molecule species Pr are eventually produced within t time units. This expressed in CSL as $\mathscr{P}_{=?}(\Diamond^{[0,t]}Pr = 4)$.

WGC A wireless group communication protocol analysed by Massink et al. [MLK04]. It is a variant of a subset of the IEEE 802.11 standard describing a subnet consisting of $N = 4$ wireless stations and an access point. The number of consecutive losses of a message transmitted through the network is described by the omission degree. The higher the omission degree, the bigger the state space. In our runs, we took $OD = 32$, becoming the largest model in the selection. The measure of interest is the probability that a message sent out by the access point is not received by any station within a given timeframe t. This is expressed in CSL as $\mathscr{P}_{=?}(\Diamond^{[0,t]} \mathit{fail})$.

From the above case studies, the first three models are part of PRISM's repository of case studies. The WGC and ER models are not part of the official PRISM repository, but are expressed in PRISM and afterwards automatically converted to MRMC's file format using PRISM's built-in converter. An overview of the models metrics can be found in Table 8.3. The stiffness is defined as the ratio of the largest rate to the smallest rate in the CTMC.

Table 8.3: Model properties of the case studies.

Model	States	Transitions	Stiffness
CSPS	3072	14848	1600
TQN	861	2859	400
PTP	1024	5121	0.5
ER	4011	11431	4000000
WGC	1329669	9624713	6164

8.4.4 Results

The results of the runs for all three configurations are described in Table 8.4 and Table 8.5. The timebound column describes the different upper timebounds used in the CSL property. The #terms column describes the number of terms in the series needed for uniformisation to meet the error level 10^{-6}. The column m describes the Krylov subspace dimension needed to meet the error level 10^{-6}. The memory column is the peak memory consumption measured using Linux's processes interface. The time column is the running time for a particular configuration. The probability column shows the computed probabilities by both algorithms. For all three configurations, the computed probabilities were exactly the same (and within the error level of 10^{-6}).

Table 8.4: Verification times and memory consumption of the non-stiff models.

Model	Time-bound	Terms		m	Memory (KB)			Time (ms)			Probability
		UNI	UNI-S	KRY	UNI	UNI-S	KRY	UNI	UNI-S	KRY	
CSPS	10	545	653	109	2944	2992	7256	190	360	2677	0.6524983
	20	769	922	132	2944	2992	9072	340	680	5633	0.8982785
	30	941	1129	147	2940	2992	10104	490	980	9158	0.9708183
	40	1086	1303	155	2944	2992	10656	640	1280	11249	0.9916387
	50	1214	1456	157	2940	2988	10928	780	1580	11933	0.9976044
	60	1330	1595	162	2944	2992	11352	940	1860	14133	0.9993137
	70	1436	1722	162	2940	2992	11348	1070	2170	13992	0.9998034
	80	1535	1841	162	2944	2988	11352	1230	2470	13811	0.9999437
	90	1627	1952	162	2944	2992	11352	1380	2760	13651	0.9999839
	100	1715	2058	162	2940	2992	11352	1530	3060	14038	0.9999954
TQN	0.02	144	173	12	92	96	92	0	0	5	0
	0.07	149	178	21	96	92	96	0	0	16	1.7e-06
	0.12	153	182	26	92	96	96	0	0	25	0.0019782
	0.17	157	186	29	96	92	96	0	0	32	0.0550075
	0.22	161	190	31	96	92	92	0	0	39	0.2875958
	0.27	166	195	32	92	96	92	0	0	41	0.6267612
	0.32	170	199	34	96	92	96	0	0	51	0.8643245
	0.37	173	203	35	96	92	92	0	0	52	0.9638449
	0.42	175	208	35	96	96	92	10	0	52	0.992505
	0.47	177	212	39	96	96	96	0	0	67	0.9987298

(continued on next page)

(continued from previous page)

Model	Time-bound	Terms		m	Memory (KB)			Time (ms)			Probability
		UNI	UNI-S	KRY	UNI	UNI-S	KRY	UNI	UNI-S	KRY	
PTP	1	163	192	20	96	96	92	10	10	18	0.3892596
	2	177	212	23	96	96	92	10	20	23	0.9055015
	3	184	220	25	96	96	96	10	20	28	0.987485
	4	190	228	25	92	96	92	10	20	28	0.9983193
	5	195	234	26	92	96	96	20	20	31	0.9997729
	6	200	240	26	92	96	96	10	20	30	0.9999693
	7	204	245	26	96	96	96	20	20	31	0.9999958
	8	208	250	26	96	92	92	20	20	32	0.9999994
	9	212	254	27	96	96	96	20	20	33	0.9999999
	10	216	259	28	96	96	92	20	20	36	1

Non-Stiff Models

Considering the stiffness ratios in Table 8.3, we classified the models PTP, TQN and CSPS as non-stiff. These models have been well-studied using uniformisation-based Markov analysis tools. The results of these case studies are outlined in Table 8.4. It shows that the Krylov algorithm is generally slower than the uniformisation-based algorithm for non-stiff models. This observation highlights a class of models for which uniformization is known to work well: non-stiff to mildly-stiff sparse models. The uniformisation rates needed for these models are small and thus the number of terms needed by uniformisation is small. Note that the increase of the upper time bound directly correlates with the increase in number of terms. Also, the number of terms of UNI-S is bigger than that of UNI. This is due to the steady state detection, which requires tighter left and right truncation points for determining the steady state correctly [KZ06].

Besides uniformisation's well explainable performance characteristics for non-stiff to mildly stiff models, the Krylov-based method has a higher constant cost due to the Arnoldi iteration which computes a dense projection and Hessenberg matrix. Furthermore, despite the small size of the non-stiff models, a relatively large —though absolutely measured small— subspace dimension is needed to meet the desired error level.

Stiff Models

The models ER and WGC are considered to be stiff. The results for these case studies are outlined in Table 8.5. Note that the probabilities for the WGC case study are all zero. This is expected since we chose a high omission degree (32) in order to increase the state space size. High omission degrees significantly reduce the probability that the message is not received (cf. [Zap08]).

The results show that for these models, the Krylov-based method is an order of magnitude faster than uniformisation. This can be seen in Figures 8.1 and 8.2 which plot verification times (in ms) against the time bound of the CSL property. When uniformisation is performed with steady-state detection, Krylov's performance gain over uniformisation even increases. The figures show that the running times of uniformisation (with and without steady-state detection) are obviously linear. The running times for the Krylov runs appear to be constant. A linear regression however showed that the slope of Krylov's running times are also linear, though with a very slow slope, whereas the slopes of uniformisation are significantly higher.

These performance characteristics are explainable akin to the non-stiff models. Uniformization is sensitive to the uniformization rate and the upper time bound. The high stiffness is the direct cause for the former and causes uniformization to compute a significant amount of terms in order to satisfy the desired error level. Larger upper time bounds additionally increase that amount of terms. The Krylov-based method does not suffer much from the stiffness, as the infin-

Table 8.5: Verification times and memory consumption of the stiff models.

Model	Time-bound	Terms		m	Memory (KB)			Time (ms)			Probability
		UNI	UNI-S	KRY	UNI	UNI-S	KRY	UNI	UNI-S	KRY	
ER	100	7638	9166	51	3108	3184	5653	490	930	372	0.1408622
	200	10801	12960	54	3136	3212	6600	950	1840	442	0.5621672
	300	13227	15872	55	3152	3236	6468	1400	2740	470	0.8457958
	400	15273	18326	55	3172	3128	6664	1880	3640	454	0.9563306
	500	17075	20489	56	3056	3152	5652	2340	4550	455	0.9892358
	600	18704	22444	56	3068	3160	6600	2770	5420	484	0.9975874
	700	20203	24243	56	3084	3180	6464	3260	6340	488	0.9994953
	800	21597	25916	56	3096	3192	6464	3720	7220	492	0.9998998
	900	22907	27488	59	3100	3204	6600	4170	8150	556	0.9999809
	1000	24146	28975	60	3112	3126	7164	4640	9030	581	0.9999965
WGC	10000	578	693	33	392884	412644	1027892	171490	342550	109549	0
	20000	816	979	34	392884	417136	1048668	325930	650460	116335	0
	30000	999	1197	35	392888	417124	1069448	477890	953350	123042	0
	40000	1153	1382	35	392884	417128	1090224	628630	1253070	123357	0
	50000	1288	1545	36	392888	417128	1090220	778280	1551270	130295	0
	60000	1411	1692	36	392888	417132	1090220	927990	1848150	130355	0
	70000	1523	1827	36	392888	417128	1090220	1076650	2144230	130271	0
	80000	1629	1953	36	392888	417128	1111000	1225090	2439050	134827	0
	90000	1727	2071	36	392888	417124	1111000	1372840	2734100	130291	0
	100000	1820	2184	37	392884	412648	1110996	1519720	3026840	145825	0

itesimal generator matrix can be approximated accurately by a small Hessenberg matrix, and thus the Krylov technique terminates quickly. This compensates for the relatively high costs of the Arnoldi iteration.

Peak Memory Consumption

Both the memory columns in Table 8.5 and Table 8.4 indicate that uniformisation has a clear advantage over the Krylov-based method when it comes to peak memory consumption. This can be explained to the storage of the dense projection matrix which is of size $m \times dim(A)$. Krylov's memory consumption increases for larger time bounds, although the increase is slow.

8.5 Discussion

We showed using Schwerdtfeger's formula how the Krylov-based method is well suited for computing the transient distribution and present it as a performant alternative to uniformisation. The experimental results on a selection of five case studies from literature revealed that the Krylov-based implementation is an order of magnitude faster than uniformisation on stiff models. This comes at the cost of increased memory consumption. If running time is the bottleneck, and if the model is stiff, our observations indicate that for time-bounded reachability properties a Krylov-based method is preferable over the commonly used uniformisation.

Several other works have tackled stiffness from different perspectives. In [MS94], an alternative method called adaptive uniformisation is presented. It essentially reduces the state space of a CTMC by slicing away "in-active" states and keeping the active states. The latter is defined as the states that are reachable within a predefined number, n say, of steps in the uniformised matrix. The usefulness of the method depends on the chosen n and the model itself. This is in contrast to the Krylov-based approach, which does not need additional input parameters. The advantage of adaptive uniformisation, however, is that an a priori error estimate can be given, as for standard uniformisation. Adaptive uniformisation has recently been combined with abstraction techniques [HMW09]. Another work is called "*Uniformisation Power*" [AM93]. It is an optimization over Jensen's uniformisation to increase the numerical stability and performance. It essentially performs scaling and squaring by subdividing the time interval to $t' = \frac{t}{2^n}$ and modifying equation 8.2 to calculate $\pi(t)$ as $\pi(t) = \mathbf{\Pi}^n$, with $\mathbf{\Pi} = (\sum_{k=0}^{\infty} e^{-\Lambda t'} \frac{(\Lambda t')^k}{k!} \mathbf{P}^k) \cdot \pi(0)$. The advantage is that it requires much smaller steps for large Λ than standard uniformisation. The drawback is that matrix $\mathbf{\Pi}$ is dense, and thus the amount of memory required is excessive. Also related, but not in the uniformisation-paradigm is the work by Carrasco [Car03]. Their technique, called regenerative randomization with Laplace transform inversion operates over Markov reward models. The truncated transformed model obtained in this regenerative method is solved using

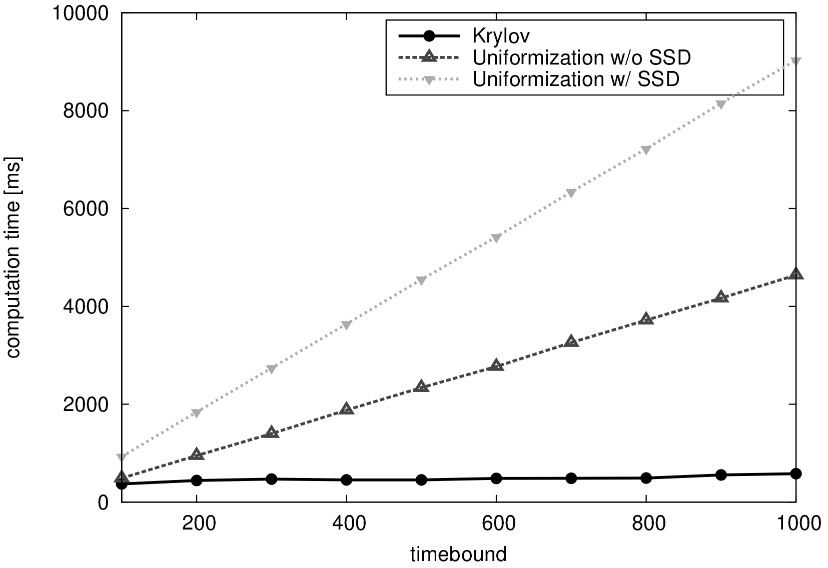


Figure 8.1: Verification times of $\mathscr{P}_{=?}(\Diamond^{[0,t]} Pr = 4)$ with increasing timebounds t on the ER model.

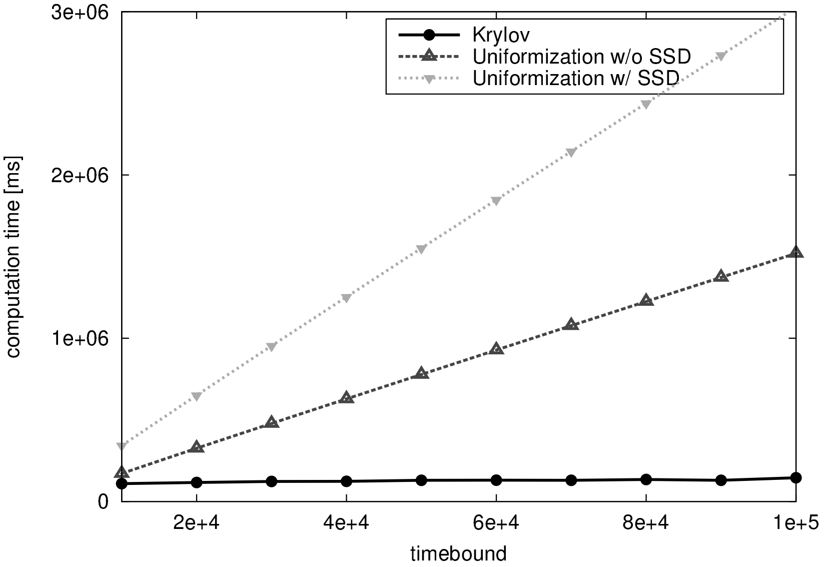


Figure 8.2: Verification times of $\mathscr{P}_{=?}(\Diamond^{[0,t]} \mathit{fail})$ with increasing timebounds t on the WGC model.

a Laplace transform inversion algorithm instead. The main difficulty in this technique is to find appropriate regeneration points. This is doable for certain classes of models, such as failure-repair models, but in general is a non-trivial issue. For stiff models with absorbing states (as for time-bounded reachability properties), this technique outperforms standard uniformisation when the model is not very large. We also explored the use of ODE solvers to deal with stiffness. Numerous techniques and optimizations have been developed to handle stiff ODEs, which is a well-known phenomenon in various scientific disciplines. The available ODE solver libraries are generally extensively tested and understood. The drawback is that ODE solvers are typically developed for a general class of problems and thus not optimized for the Markovian case. Nevertheless, we briefly experimented with Intel's ODE solver package by interfacing it with MRMC. Unfortunately the package did not expose sufficient information about the induced error, rendering its application for the transient unsatisfied.

Closer to our own work is that by Sidje [Sid98]. He wrote a toolkit called Expokit which computes matrix exponentials using a variety of methods, including one using Krylov subspaces. It also comes with an optimized version for computing the transient of Markov chains. The error bounds used are based on the assumption that the elements of the transient vector sums up to 1. This however does not hold for CSL model checking because the initial vectors are interpreted differently, causing the elements not necessarily sum up to one. The toolkit also comes with a time-stepping scheme of the matrix exponential which subdivides the time bound and computes the transient in steps. Our implementation in MRMC is inspired by Expokit's MatLab code.

The field of Krylov-based subspaces is relatively young, and though its applications are spreading fast, there are several gaps open for further study. An open question is the relation of stiffness and the eigenvalues spectrum of a CTMC. Our debugging observations hint to a correlation between the stiffness and the way eigenvalues are spread in their spectrum. For the non-stiff models in the selected case studies, the eigenvalues are homogeneously spread across the spectrum. There is however no theoretical evidence to claim that this generally holds. There is a report however that a high clustering of eigenvalues is beneficial for the Krylov-based method [Wei94] and this would back our experimental data. Further study is required to fully understand this. An interesting direction of future work is improving the error bounds. Initially we experimented with several a priori error bounds developed by Hochbruck et al. [HL97], but found these error bounds too conservative when compared to Saad's a posteriori error estimates. Hochbruck et al.'s bounds are also expensive to compute because they are based on the numerical range of the input matrix. It is however an open question whether Saad's a posteriori error estimate (see Equation 8.5) is a bounded over-approximation of the real error (see Equation 8.4). The study of improved a priori error bounds for Krylov-based matrix exponentials is however an active field. Advancements from there are directly applicable to the computation of the transient. It would be desirable to have a cheap, yet reliable bound, enabling us to automatically determine the required subspace dimension in advance. In the mean time, a posteriori error estimates are the best choice for practical applications. Then another point for future study is the trend towards parallelised architectures. The Krylov-based method is highly amenable to such architectures because it is mostly based on matrix-vector multiplications. Data-parallelism in graphics cards can be exploited to achieve a significant performance gain. This gain would be especially visible for large models. For DTMCs, such architectures have already been exploited with positive results [BES09].

9

Conclusion

Little more than four years ago, Joost-Pieter recruited me to govern the technical realisation of COMPASS. Back then, just before I started, I told him that innovation-projects of such nature typically *fail*, but that I shall try defying statistics. I did not comprehend the full scope of the COMPASS project back then. In retrospect, it is slightly ironic that COMPASS is now employed to investigate causes and effects of failures, be it that of spacecraft. This is especially demonstrated by our case study in Chapter 6. Requests from our American acquaintances to export our work outside the member states of the European Space Agency also mark the need for technologies like COMPASS.

Within the esa5458 archive of COMPASS, one can trace all information of the project back to the agency's original statement of work. It contains all discussions, technical specifications, analyses, reports, presentations, publications and the source code itself. These are however not public. For our scientific peers, there is this dissertation. A significant part of it, Chapters 3 to 5, describes the scientific aspects of COMPASS in an integral and coherent manner. And the novel approaches we developed for COMPASS are compared to existing works.

During the COMPASS project, I found myself mostly on the academic side. Within the project we formed a close cooperation with industry (Thales Alenia Space) and government (ESA). By this, I got an unique insight on the space practice of developing embedded systems and its software. Its demanding requirements bring along a dimension of rigour and completeness that would only hesitantly be pushed for during the development of terrestrial systems. The challenges faced within space systems engineering calls for the benefits of formal methods. When, after COMPASS, I was offered the opportunity to continue researching such systems, partly within ESA and partly at the RWTH, I took it without hesitation.

Accumulatively, I have nearly spent a full year at ESA's European Space Research and Technology Centre. The unofficial time spent is even a tad longer. Through discussions with ESA specialists of various engineering disciplines, and an in-house training on space systems engineering and its verification and validation aspects, I developed a deeper insight in to the processes and aspects that play a role during the technical realisation of a space mission. Chapter 2 reflects on this, be it in a summarised and tailored way. I furthermore studied technical documents from several missions in development. Having such information within my reach shaped much of my view of how *real* industrial-scale systems are designed and work. From one of those missions, we crafted the case study whose result is Chapter 6. It is, to our knowledge, the largest case study of a system-level model that has been analysed using formal methods. The study made clear what the current state of formal methods can do, and on which points improvements are desired.

Before we ran the case study, we were already aware of the potential issue of the state space explosion. The case study confirmed it. It is especially an issue with diagnosability analysis. I investigated it for more than a year. There is a plethora of techniques that tackle it, but they are typically only applicable under peculiar conditions, and as such are no generic solutions. After many attempts, and setbacks, I developed a theoretical solution based on interpolation. It is not only applicable to SLIM, but to many other component-oriented languages that have an encoding as a bounded model checking formula. This solution is described in Chapter 7.

Chapter 8 is a result that was initially triggered by an observation of our assigned ESA technical officer, who remarked that failure rates of (space) components could have large disparities. This is stiffness. We noticed severe performance degradation on stiff models when we used uniformisation to compute the transient probabilities. During and after the COMPASS project, we experimented with alternative approaches. Out of the box ordinary differential equation solvers were are first attempt, but did not work out as the good solvers do not provide much insight on controlling stiffness against numerical instability. A colleague and my office mate, Alexandru, suggested a book chapter as an inspirational read. The chapter refers to Krylov subspace methods as one of the latest and exciting advances in linear algebra. A subsequent gathering with scientists specialised in Krylov subspace methods at the Technical University of Delft made it clear to me that they can be used to compute the transient. Our implementation and experiments delivered on the promise, showing that the Krylov subspace method performs better on stiff models than uniformisation.

For the past five years, I investigated both state of the art formal methods and space system software engineering. Their combination is a synergy from which matters for near-future improvement became better identifiable.

First of all, we need a broad encompassing formal semantics and model composability constructs that cover interrelations of domains that are typically separated. In COMPASS we married the design of system software with that of safety & dependability by providing the notion of Markovian transitions in the formal semantics and the employment of model extension for merging artefacts from the safety & dependability domain with the system software domain. I argue that more engineering domains, like security and cost engineering, need to be included. It requires a deep understanding of interrelations between them. The benefit is that more engineering analyses can be derived from the same system model, thereby strengthening the justification of formal system-level modelling.

Secondly, we need model analyses that generate intelligible artefacts that provide a profound understanding of the causality of events. Especially in the domain of fault management engineering, a clear comprehension of failure events and their propagations is crucial for an effective fault management design. Fault trees are currently the artefact of choice. At the moment, COMPASS generates the causality relation between a top-level event and sets of error events. The causally dependent nominal events in between are not synthesized. As there are many, this requires a proper notion of granularity for a sensible presentation to the user. The benefit is an improved comprehension of the causality of non-nominal events and nominal events, and not just either of them in isolation.

We furthermore need techniques that make formal analyses more tractable for industrial practice. I consider our work on component-oriented interpolation as a first step. It aggressively abstracts the state space while preserving good convergence properties, as our experiments indicate to us. I think the technique can be further extended to support a broader class of properties, like safety and even liveness properties. It needs to be investigated whether the approach is also amenable for the SAT-encoding of timed reachability in a network of timed automata. Besides increasing the range of supported logics and models, it also requires further investigation how to exploit compositional reasoning for analyses that depend on reachability of the state space, like fault tree generation and diagnosability. It is only then when theoretical improvements become more visible to industrial practice.

Last but not least, we need to run, perhaps continually, case studies of industrial relevance with state of the art tools. Preferable concurrently with the ongoing engineering process itself. The experience gained is of tremendous value enabling a clear technology strategy through the wide-range of developments in formal methods. By placing priority and focus on the pressing methods, they receive the attention needed to mature rapidly to a technology readiness level that is desired to tackle engineering issues in industrial practice.

A

Sensor-Filter Model

The sensor-filter model is a data acquisition model originally developed by Thomas Noll and modified by the author of this dissertation. It is used as a running example for demonstrating COMPASS analyses. It consists of a redundant sensor bank, whose signal is filtered by a redundant filter bank. Both the sensor and the filter may fail, in which case a switch is made from one to the other. This logic is captured by a fault detection, isolation and recovery monitor. Example fault injections for this model can be found in Section 4.1.3. Example properties can be found in Section 4.2.3.

A.1 Nominal Model

The nominal model consists of the overall system, i.e. the acquisition systems (see Listing A.1), the sensor bank, the filter bank and the monitor. They are represented as components in the following subsections.

A.1.1 Acquisition System

The acquisition system is the root component. It should raise an alarm when both sensors have failed or when both filters have failed. Otherwise the obtained data, represented by the outgoing data port `value` is valid.

Listing A.1: Overall system represented as the acquisition component.

```
system Acquisition
  features
    value: out data port int default 4;
    alarmS: out data port bool default false observable;
    alarmF: out data port bool default false observable;
end Acquisition;
```

```
system implementation Acquisition.Impl
  subcomponents
    sensors: system Sensors accesses mybus;
    filters: system Filters accesses mybus;
    monitor: system Monitor accesses mybus;
    mybus: bus MyBus;
  connections
    data port sensors.output -> filters.input;
    data port filters.output -> value;
    data port sensors.output -> monitor.valueS;
    data port filters.output -> monitor.valueF;
    data port monitor.alarmS -> alarmS;
    data port monitor.alarmF -> alarmF;
    event port monitor.switchS -> sensors.switch;
    event port monitor.switchF -> filters.switch;
end Acquisition.Impl;
```

A bus component is used to relay information from one subcomponent to another within the acquisition component. This component performs no operation over the relayed data, and hence is empty. Its SLIM representation in shown in Listing A.2.

Listing A.2: An empty bus component.

```
bus MyBus
end MyBus;
bus implementation MyBus.Impl
end MyBus.Impl;
```

A.1.2 Sensors

The sensor bank is represented by the component Sensors.Impl. It consists of two sensors of type Sensor (cf. Listing A.4). Either sensor is active in the modes Primary or Backup. A switch is made from the primary one to the secondary one upon an incoming switch event.

Listing A.3: The sensor bank.

```
system Sensors
  features
    output: out data port int default 1;
    switch: in event port;
end Sensors;

system implementation Sensors.Impl
```

```
  subcomponents
    sensor1: device Sensor in modes (Primary);
    sensor2: device Sensor in modes (Backup);
  connections
    data port sensor1.output -> output in modes (Primary);
    data port sensor2.output -> output in modes (Backup);
  modes
    Primary: activation mode;
    Backup: mode;
  transitions
    Primary -[switch]-> Backup;
end Sensors.Impl;
```

A single sensor simply increases its output by 1 as long as the output is smaller than 5. It does not represent a real sensor, but this behaviour suffices for demonstration purposes.

Listing A.4: A single sensor.

```
device Sensor
  features
    output: out data port int default 1;
end Sensor;

device implementation Sensor.Impl
  modes
    Cycle: activation mode;
  transitions
    Cycle -[when output < 5 then output := output + 1]-> Cycle;
end Sensor.Impl;
```

A.1.3 Filters

The filter bank is similarly structured as a sensor bank: it consists of two filters which are disjunctively active in the Primary and Backup mode. A switch between modes occurs upon a switch event.

Listing A.5: The filter bank.

```
system Filters
  features
    input: in data port int default 1;
    output: out data port int default 2;
    switch: in event port;
end Filters;
```

```
system implementation Filters.Impl
  subcomponents
    filter1: device Filter in modes (Primary);
    filter2: device Filter in modes (Backup);
  connections
    data port input -> filter1.input in modes (Primary);
    data port input -> filter2.input in modes (Backup);
    data port filter1.output -> output in modes (Primary);
    data port filter2.output -> output in modes (Backup);
  modes
    Primary: activation mode;
    Backup: mode;
  transitions
    Primary -[switch]-> Backup;
end Filters.Impl;
```

A single filter simply multiplies its input by 2. Akin to the sensor, it does not represent a real filter. This behaviour suffices for demonstration purposes.

Listing A.6: A single filter.

```
device Filter
  features
    input: in data port int default 1;
    output: out data port int default 2;
end Filter;

device implementation Filter.Impl
flows
    output := input *2;
end Filter.Impl;
```

A.1.4 Monitor

The monitor is responsible for fault detection, isolation and recovery. There are two detection criteria, namely that the value drops to 0, or when the value of the sensors is above 5. Both indicate a fault. Depending on which one occurred, or possibly both, appropriate switch events are sent out. In case the redundancy has failed, the appropriate alarms are raised.

Listing A.7: FDIR monitor.

```
fdir system Monitor
  features
```

```
    valueS: in data port int default 0;
    valueF: in data port int default 0;
    switchS: out event port;
    switchF: out event port;
    alarmS : out data port bool default false;
    alarmF : out data port bool default false;
end Monitor;

fdir system implementation Monitor.Impl
  modes
    OK: activation mode;
    FailS: mode;
    FailF: mode;
    FailSF: mode;
  transitions
    OK -[switchF when valueF = 0]-> FailF;
    OK -[switchS when valueS > 5]-> FailS;
    FailF -[switchS when valueS > 5
                   then alarmF := valueF = 0]-> FailSF;
    FailF -[when valueF = 0 then alarmF := true]-> FailF;
    FailS -[switchF when valueF = 0
                   then alarmS := valueS > 5]-> FailSF;
    FailS -[when valueS > 5 then alarmS := true]-> FailS;
    FailSF -[when valueF = 0
             then alarmF := true; alarmS := valueS > 5]-> FailSF;
    FailSF -[when valueS > 5
             then alarmS := true; alarmF := valueF = 0]-> FailSF;
end Monitor.Impl;
```

A.2 Error Model

In the sensor-filter model, we consider two possible errors, namely those occurring in the sensor and those occurring in the filter. Their erroneous behaviours are described in the following subsections.

A.2.1 Sensor Errors

For demonstration purposes, a sensor is initially OK after which it can immediately die. It can also drift first before it dies. The rates of dying and drifting do not represent any real erroneous behaviour.

Listing A.8: Error model of a sensor.

```
error model SensorFailures
features
    OK: initial state;
    Drifted: error state;
    Dead: error state;
end SensorFailures;

error model implementation SensorFailures.Impl
events
    drift: error event occurrence poisson 0.083;
    die: error event occurrence poisson 0.00001;
    dieByDrift: error event occurrence poisson 0.00015;
transitions
    OK -[ die ]-> Dead;
    OK -[ drift ]-> Drifted;
    Drifted -[ dieByDrift ]-> Dead;
end SensorFailures.Impl;
```

A.2.2 Filter Errors

Akin to the sensor, a filter is initially OK after which it can die. It can also degrade first before it dies.

Listing A.9: Error model of a filter.

```
error model FilterFailures
features
    OK: initial state;
    Degraded: error state;
    Dead: error state;
end FilterFailures;
error model implementation FilterFailures.Impl
events
    die: error event occurrence poisson 0.007;
    degrade: error event occurrence poisson 0.051;
transitions
    OK -[ die ]-> Dead;
    OK -[ degrade ]-> Degraded;
    Degraded -[ die ]-> Dead;
end FilterFailures.Impl;
```

Standards, Handbooks & Manuals

[DoDI 5000.02] *Operation of the Defense Acquisition System*. DoDI 5000.02. Department of Defense, United States of America, Dec. 2008 (cit. on p. 17).

[DTM-09-025] *Space Systems Acquisition Policy*. DTM-09-025. Department of Defense, United States of America, Oct. 2010 (cit. on p. 17).

[ECSS-E-10-03A] *Testing*. ECSS-E-10-03A. ESA Requirements and Standards Division, Noordwijk, Netherlands, Feb. 2002 (cit. on p. 16).

[ECSS-E-HB-10-02A] *Verification Guidelines*. ECSS-E-HB-10-02A. ESA Requirements and Standards Division, Noordwijk, Netherlands, Dec. 2010 (cit. on pp. 14, 16).

[ECSS-E-HB-60A] *Control Engineering Handbook*. ECSS-E-HB-60A. ESA Requirements and Standards Division, Noordwijk, Netherlands, Dec. 2010 (cit. on p. 27).

[ECSS-E-ST-10-02C] *Verification*. ECSS-E-ST-10-02C. ESA Requirements and Standards Division, Noordwijk, Netherlands, Mar. 2009 (cit. on pp. 15, 16).

[ECSS-E-ST-10C] *System Engineering General Requirements*. ECSS-E-ST-10C. ESA Requirements and Standards Division, Noordwijk, Netherlands, Mar. 2009 (cit. on p. 27).

[ECSS-E-ST-20C] *Electrical and Electronic*. ECSS-E-ST-20C. ESA Requirements and Standards Division, Noordwijk, Netherlands, July 2008 (cit. on p. 27).

[ECSS-E-ST-40C] *Software*. ECSS-E-ST-40C. ESA Requirements and Standards Division, Noordwijk, Netherlands, Mar. 2009 (cit. on pp. 18, 19, 21, 23).

[ECSS-E-ST-70C] *Ground Systems and Operations*. ECSS-E-ST-70C. ESA Requirements and Standards Division, Noordwijk, Netherlands, July 2008 (cit. on p. 27).

[ECSS-E-TM-10-21A] *System Modelling and Simulation*. ECSS-E-TM-10-21A. ESA Requirements and Standards Division, Noordwijk, Netherlands, Apr. 2010 (cit. on p. 27).

[ECSS-M-ST-10C] *Project Planning and Implementation*. ECSS-M-ST-10C Rev. 1. ESA Requirements and Standards Division, Noordwijk, Netherlands, Mar. 2009 (cit. on pp. 9, 10).

[ECSS-P-001B] *Glossary of Terms*. ECSS-P-001B. ESA Requirements and Standards Division, Noordwijk, Netherlands, July 2004 (cit. on p. 12).

[ECSS-Q-HB-80-01A] *Reuse of Existing Software*. ECSS-Q-HB-80-01A. ESA Requirements and Standards Division, Noordwijk, Netherlands, Dec. 2011 (cit. on p. 23).

[ECSS-Q-HB-80-02] *Software Process Assessment and Improvement*. ECSS-Q-HB-80-02. ESA Requirements and Standards Division, Noordwijk, Netherlands, Oct. 2010 (cit. on p. 23).

[ECSS-Q-HB-80-03A] *Software Dependability and Safety*. ECSS-Q-HB-80-03A. ESA Requirements and Standards Division, Noordwijk, Netherlands, Jan. 2012 (cit. on pp. 23, 27).

[ECSS-Q-HB-80-04A] *Software Metrication Programme Definition and Implementation*. ECSS-Q-HB-80-04A. ESA Requirements and Standards Division, Noordwijk, Netherlands, Mar. 2011 (cit. on p. 27).

[ECSS-Q-ST-30-02C] *Failure Modes, Effects (and Criticality) Analysis*. ECSS-Q-ST-30-02C. ESA Requirements and Standards Division, Noordwijk, Netherlands, Mar. 2009 (cit. on p. 16).

[ECSS-Q-ST-30-09C] *Availability Analysis*. ECSS-Q-ST-30-09C. ESA Requirements and Standards Division, Noordwijk, Netherlands, July 2008 (cit. on p. 16).

[ECSS-Q-ST-30C] *Dependability*. ECSS-Q-ST-30C. ESA Requirements and Standards Division, Noordwijk, Netherlands, Mar. 2009 (cit. on p. 27).

[ECSS-Q-ST-40-12C] *Fault Tree Analysis*. ECSS-Q-ST-40-12C. ESA Requirements and Standards Division, Noordwijk, Netherlands, July 2008 (cit. on pp. 16, 70).

[ECSS-Q-ST-40C] *Safety*. ECSS-Q-ST-40C. ESA Requirements and Standards Division, Noordwijk, Netherlands, Mar. 2009 (cit. on p. 17).

[ECSS-Q-ST-80C] *Software Product Assurance*. ECSS-Q-ST-80C. ESA Requirements and Standards Division, Noordwijk, Netherlands, Mar. 2009 (cit. on p. 23).

[MIL-HDBK-217F] *Reliability Prediction of Electronic Equipment*. MIL-HDBK-217F. Department of Defense, United States of America, Dec. 1991 (cit. on p. 74).

[MIL-HDBK-338B] *Electronic Reliability Design Handbook*. MIL-HDBK-338B. Department of Defense, United States of America, Oct. 1998 (cit. on p. 87).

[NASA-GB-8719.13] *NASA Software Safety Guidebook*. NASA-GB-8719.13. NASA, Mar. 2004 (cit. on p. 23).

[NASA-HDBK-1002] *NASA Fault Management Handbook*. Version 1 (Draft). NASA-HDBK-1002. NASA, Jan. 2011 (cit. on p. 27).

[NASA/SP-2007-6105] *NASA Systems Engineering Handbook*. NASA/SP-2007-6105 Rev. 1. NASA, Dec. 2007 (cit. on p. 17).

[NASA/SP-2010-580] *NASA System Safety Handbook*. NASA/SP-2010-580. NASA, Nov. 2011 (cit. on pp. 17, 70).

[NASA/SP-2011-3421] *Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners*. NASA/SP-2011-3421. NASA, Dec. 2011 (cit. on p. 17).

[NPR 7123.1A] *NASA Systems Engineering Processes and Requirements*. NPR 7123.1A. NASA, Mar. 2007 (cit. on p. 17).

[NPR 7150.2A] *NASA Software Engineering Requirements*. NASA, Nov. 2009 (cit. on p. 23).

[SAE-AS5506] *Architecture, Analysis and Design Language*. AS5506. Society of Automotive Engineers, Nov. 2004 (cit. on p. 28).

[SAE-AS5506/1] *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1*. AS5506/1. Society of Automotive Engineers, June 2006 (cit. on pp. 33, 50).

[SAE-AS5506/2] *SAE Architecture Analysis and Design Language (AADL) Annex Volume 2*. AS5506/2. Society of Automotive Engineers, Jan. 2011 (cit. on p. 51).

Publications

[Aba+09] A. Abate, A. D'Innocenzo, M.D.D. Benedetto and S. Sastry. 'Understanding deadlock and livelock behaviors in Hybrid Control Systems'. In: *Nonlinear Analysis: Hybrid Systems* 3.2 (2009), pp. 150–162 (cit. on p. 69).

[ACD93] R. Alur, C. Courcoubetis and D. Dill. 'Model-Checking in Dense Real-Time'. In: *Information and Computation* 104.1 (1993), pp. 2–34 (cit. on p. 69).

[ADBNR09] N.H. Aan De Brugh, V.Y. Nguyen and T.C. Ruys. 'MoonWalker: Verification of .NET Programs'. In: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by S. Kowalewski and A. Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 170–173 (cit. on p. 7).

[AM93] H. Abdallah and R. Marie. 'The Uniformized Power Method for Transient Solutions of Markov Processes'. In: *Computers & Operations Research* 20.5 (1993), pp. 515–526 (cit. on p. 149).

[Aud+02] G. Audemard, A. Cimatti, A. Kornilowicz and R. Sebastiani. 'Bounded Model Checking for Timed Systems'. In: *Proceedings of the 22nd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*. Ed. by D. Peled and M.Y. Vardi. Vol. 2529. Lecture Notes in Computer Science. Springer, 2002, pp. 243–259 (cit. on p. 108).

[Aud+05] G. Audemard, M. Bozzano, A. Cimatti and R. Sebastiani. 'Verifying Industrial Hybrid Systems with MathSAT'. In: *Proceedings of the 2nd International Workshop on Bounded Model Checking (BMC)*. Ed. by A. Biere and O. Strichman. Vol. 119. Electronic Notes in Theoretical Computer Science 2. Elsevier, 2005, pp. 17–32 (cit. on p. 104).

[Bae+11] K. Bae, P.C. Ölveczky, A. Al-Nayeem and J. Meseguer. 'Synchronous AADL and Its Formal Analysis in Real-Time Maude'. In: *Proceedings of the 13th International Conference on Formal Engineering Methods*

(ICFEM). Ed. by S. Qin and Z. Qiu. Vol. 6991. Lecture Notes in Computer Science. Springer, 2011, pp. 651–667 (cit. on p. 54).

[Bai+03] C. Baier, B. Haverkort, H. Hermanns and J.-P. Katoen. 'Model Checking Algorithms for Continuous-Time Markov Chains'. In: *Transactions on Software Engineering* 29.6 (2003), pp. 524–541 (cit. on pp. 74, 89, 136).

[BCS07] H. Boudali, P. Crouzen and M. Stoelinga. 'Dynamic Fault Tree Analysis Using Input/Output Interactive Markov Chains'. In: *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2007, pp. 708–717 (cit. on pp. 70, 74).

[BCT07] M. Bozzano, A. Cimatti and F. Tapparo. 'Symbolic fault tree analysis for reactive systems'. In: *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis (ATVA)*. Ed. by K.S. Namjoshi, T. Yoneda, T. Higashino and Y. Okamura. Vol. 4762. Lecture Notes in Computer Science. Springer, 2007, pp. 162–176 (cit. on p. 71).

[Ber+09] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S.D. Zilio, M. Filali and F. Vernadat. 'Formal Verification of AADL Specifications in the Topcased Environment'. In: *Proceedings of the 14th International Conference on Reliable Software Technologies (RST)*. Ed. by F. Kordon and Y. Kermarrec. Vol. 5570. Lecture Notes in Computer Science. Springer, 2009, pp. 207–221 (cit. on p. 54).

[BES09] D. Bosnacki, S. Edelkamp and D. Sulewski. 'Efficient Probabilistic Model Checking on General Purpose Graphics Processors'. In: *Proceedings of the 16th International Conference on Model Checking Software (SPIN)*. Ed. by C.S. Pasareanu. Vol. 5578. Lecture Notes in Computer Science. Springer, 2009, pp. 32–49 (cit. on p. 152).

[BG06] H. Bowman and R. Gómez. 'How to Stop Time Stopping'. In: *Formal Aspects of Computing* 18.4 (2006), pp. 459–493 (cit. on p. 69).

[BGG03] H. Barringer, D. Giannakopoulou and D. Giannakopoulou. 'Proof Rules for Automated Compositional Verification through Learning'. In: *Proceedings of the 2nd Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*. Ed. by M. Barnett, S.H. Edwards, D. Giannakopoulou and G.T. Leavens. TR #03-11. Iowa State University, 2003, pp. 14–21 (cit. on p. 133).

[Bie+99] A. Biere, A. Cimatti, E.M. Clarke and Y. Zhu. 'Symbolic Model Checking without BDDs'. In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Ed. by R. Cleaveland. Vol. 1384. Lecture Notes in Computer Science. Springer, 1999, pp. 193–207 (cit. on pp. 66, 67, 118).

[Bit+11a] B. Bittner, M. Bozzano, A. Cimatti and X. Olive. 'Symbolic Synthesis of Observability Requirements for Diagnosability'. In: *Proceedings of 11th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*. European Space Agency, 2011 (cit. on p. 84).

[Bit+11b] B. Bittner, F. Gretz, V.Y. Nguyen, T. Noll and D. Tomasoni. *Integrated Platform User Manual*. Tech. rep. RWTH Aachen University, May 2011 (cit. on p. 97).

[Bjo+11] S. Björnander, C. Seceleanu, K. Lundqvist and P. Pettersson. 'ABV - A Verifier for the Architecture Analysis and Design Language (AADL)'. In: *Proceedings of 16th International Conference on Engineering of Complex Computer Systems (ICECCS)*. Ed. by I. Perseil, K. Breitman and R. Sterritt. IEEE, 2011, pp. 355–360 (cit. on p. 54).

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008 (cit. on p. 66).

[Boz+04] M. Bozga, S. Graf, I. Ober, I. Ober and J. Sifakis. 'The IF Toolset'. In: *In the Proceedings of the 2nd School on Formal Methods for the Design of Computer, Communication and Software Systems*. Ed. by M. Bernardo and F. Corradini. Vol. 3185. Lecture Notes in Computer Science. Springer, 2004, pp. 131–132 (cit. on p. 54).

[Boz+09a] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll and M. Roveri. 'Model-Based Codesign of Critical Embedded Systems'. In: *Proceedings of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*. Ed. by S. van Baelen, T. Weigert, I. Ober and H. Espinoza. Vol. 507. CEUR, 2009, pp. 87–91 (cit. on p. 6).

[Boz+09b] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll and M. Roveri. 'The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems'. In: *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*. Ed. by B. Buth, G. Rabe and T. Seyfarth. Vol. 5775. Lecture Notes in Computer Science. Springer, 2009, pp. 173–186 (cit. on p. 6).

[Boz+09c] M. Bozzano, A. Cimatti, M. Roveri, J.-P. Katoen, V.Y. Nguyen and T. Noll. 'Codesign of Dependable Systems: a Component-Based Modeling Language'. In: *Proceedings of the 7th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2009, pp. 121–130 (cit. on p. 6).

[Boz+09d] M. Bozzano, A. Cimatti, M. Roveri, J.-P. Katoen, V.Y. Nguyen and T. Noll. 'Verification and Performance Evaluation of AADL models'. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of*

Software Engineering (ESEC/FSE). Ed. by H. van Vliet and V. Issarny. ACM, 2009, pp. 285–286 (cit. on p. 6).

[Boz+10a] M. Bozzano, R. Cavada, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll and X. Olive. 'Formal Verification and Validation of AADL Models'. In: *Proceedings of the 4th Conference on Embedded Real Time Software and Systems Conference (ERTS^2)*. AAAF & SEE. 2010 (cit. on pp. 6, 100, 110).

[Boz+10b] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, M. Roveri and R. Wimmer. 'A Model Checker for AADL'. In: *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*. Ed. by T. Touili, B. Cook and P. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 562–565 (cit. on p. 6).

[Boz+11] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll and M. Roveri. 'Safety, Dependability and Performance Analysis of Extended AADL Models'. In: *Computer Journal* 54.5 (2011), pp. 754–775 (cit. on p. 7).

[BSW06] H. Busch, W. Sandmann and V. Wolf. 'A Numerical Aggregation Algorithm for the Enzyme-Catalyzed Substrate Conversion'. In: *Proceedings of the 4th International Conference on Computational Methods in Systems Biology (CMSB)*. Ed. by C. Priami. Vol. 4210. Lecture Notes in Computer Science. Springer, 2006, pp. 298–311 (cit. on p. 143).

[Bus97] S.R. Buss. 'Propositional Proof Complexity: An Introduction'. In: *Computational Logic* (1997) (cit. on p. 118).

[BV03a] M. Bozzano and A. Villafiorita. 'Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform'. In: *In the Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP)*. Ed. by S. Anderson, M. Felici and B. Littlewood. Vol. 2788. Lecture Notes in Computer Science. Springer, 2003, pp. 49–62 (cit. on pp. 55, 71, 94).

[BV03b] M. Bozzano and A. Villafiorita. 'Integrating Fault Tree Analysis with Event Ordering Information'. In: *Proceedings of the 3th European Safety and Reliability Conference (ESREL)*. Ed. by T. Bedford and P. van Gelder. Balkema, 2003, pp. 247–254 (cit. on p. 71).

[Car03] J. Carrasco. 'Transient Analysis of Rewarded Continuous Time Markov Models by Regenerative Randomization with Laplace Transform Inversion'. In: *The Computer Journal* 46.1 (2003), pp. 84–99 (cit. on p. 149).

[Car11] S. de Carolis. 'Zuverlässigkeitsanalyse Dynamischer Fehlerbäume'. Master's thesis. RWTH Aachen University, 2011 (cit. on pp. 76, 77).

[CGP03] J.M. Cobleigh, D. Giannakopoulou and C.S. Păsăreanu. ‘Learning Assumptions for Compositional Verification’. In: *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by H. Garavel and J. Hatcliff. Vol. 2619. Lecture Notes in Computer Science. Springer, 2003, pp. 331–346 (cit. on p. 133).

[Che+09] T. Chen, T. Han, J.-P. Katoen and A. Mereacre. ‘Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications’. In: *Proceedings of the 24th Symposium on Logic in Computer Science (LICS)*. IEEE, 2009, pp. 309–318 (cit. on p. 137).

[Chk+08] M.Y. Chkouri, A. Robert, M. Bozga and J. Sifakis. ‘Translating AADL into BIP - Application to the Verification of Real-Time Systems’. In: *Proceedings of the 1st International Workshop on Model-Based Architecting and Construction of Embedded Systems (ACES-MB)*. Ed. by S. van Baelen, I. Ober, S. Graf, M. Filali, T. Weigert and S. Gerard. Vol. 503. CEUR, 2008 (cit. on p. 54).

[Cim+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella. ‘NuSMV 2: An OpenSource Tool for Symbolic Model Checking’. In: *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Ed. by E. Brinksma and K.G. Larsen. Vol. 2404. Lecture Notes in Computer Science. Springer, 2002, pp. 359–364 (cit. on p. 94).

[CN08] A. Cohen and K.S. Namjoshi. ‘Local Proofs for Linear-Time Properties of Concurrent Programs’. In: *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*. Ed. by A. Gupta and S. Malik. Vol. 5123. Lecture Notes in Computer Science. Springer, 2008, pp. 149–161 (cit. on pp. 132, 133).

[CN09] A. Cohen and K.S. Namjoshi. ‘Local Proofs for Global Safety Properties’. In: *Formal Methods in System Design* 34.2 (2009), pp. 104–125 (cit. on pp. 132, 133).

[Cof+12] D.D. Cofer, A. Gacek, S.P. Miller, M.W. Whalen, B. LaValley and L. Sha. ‘Compositional Verification of Architectural Models’. In: *Proceedings of the 4th NASA Formal Methods Symposium (NFM)*. Ed. by A.E. Goodloe and S. Person. Vol. 7226. Lecture Notes in Computer Science. Springer, 2012, pp. 126–140 (cit. on p. 133).

[Coh+11] A. Cohen, K.S. Namjoshi, Y. Sa’ar, L.D. Zuck and K.I. Kisyova. ‘Parallelizing a Symbolic Compositional Model-Checking Algorithm’. In: *Proceedings of the 6th Haifa Verification Conference*. Ed. by S. Barner, I.G. Harris, D. Kroening and O. Raz. Vol. 6504. Lecture Notes in Computer Science. Springer, 2011, pp. 46–59 (cit. on p. 132).

[Coo71] S.A. Cook. 'The Complexity of Theorem-Proving Procedures'. In: *Proceedings of the 3th Symposium on Theory of Computing (STOC)*. Ed. by M.A. Harrison, R.B. Banerji and J.D. Ullman. ACM, 1971, pp. 151–158 (cit. on p. 116).

[CPC03] A. Cimatti, C. Pecheur and R. Cavada. 'Formal Verification of Diagnosability via Symbolic Model Checking'. In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by G. Gottlob and T. Walsh. Morgan Kaufmann, 2003, pp. 363–369 (cit. on p. 81).

[Cra57] W. Craig. 'Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory'. In: *The Journal of Symbolic Logic* 22.3 (1957), pp. 269–285 (cit. on p. 118).

[DAC99] M.B. Dwyer, G.S. Avrunin and J.C. Corbett. 'Patterns in Property Specifications for Finite-State Verification'. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE)*. Ed. by B.W. Boehm, D. Garlan and J. Kramer. ACM, 1999, pp. 411–420 (cit. on pp. 61, 64, 107).

[DGL92] I. Duff, R. Grimes and J. Lewis. *User's Guide for the Harwell-Boeing Sparse Matrix Collection*. Technical Report TR/PA/92/86. CERFACS, 1992 (cit. on p. 142).

[Dil+94] L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith and Y.S. Ramakrishna. 'A Graphical Interval Logic for Specifying Concurrent Systems'. In: *Transactions on Software Engineering and Methodology (TOSEM)* 3.2 (1994), pp. 131–165 (cit. on p. 61).

[DKN10] F. Dulat, J.-P. Katoen and V.Y. Nguyen. 'Model Checking Markov Chains using Krylov Subspace Methods: An Experience Report'. In: *Proceedings of the 7th European Performance Engineering Workshop (EPEW)*. Ed. by A. Aldini, M. Bernardo, L. Bononi and V. Cortellessa. Vol. 6977. Lecture Notes in Computer Science. Springer, 2010, pp. 115–130 (cit. on p. 6).

[DKW08] V. D'Silva, D. Kroening and G. Weissenbacher. 'A Survey of Automated Techniques for Formal Software Verification'. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (2008) (cit. on p. 126).

[dSeSG99] E. de Souza e Silva and H.R. Gail. 'Computational Probability'. In: ed. by W.K. Grassmann. Vol. 24. International Series in Operations Research & Management Science. Kluwer, 1999. Chap. Transient Solutions for Markov Chains, pp. 43–81 (cit. on pp. 135, 136).

[DSi+10] V. D'Silva, D. Kroening, M. Purandare and G. Weissenbacher. 'Interpolant Strength'. In: *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by G. Barthe and M. Hermenegildo. Lecture Notes in Computer Science. Springer, 2010, pp. 139–209 (cit. on p. 128).

[Dvo09] D.L. Dvorak. *NASA Study on Flight Software Complexity*. Tech. rep. California Institute of Technology, 2009 (cit. on p. 20).

[Ern12] B. Ern. 'Model-Based Criticality Analysis by Impact Isolation'. Master's thesis. RWTH Aachen University, 2012 (cit. on pp. 5, 57, 79, 95).

[Est+12] M.-A. Esteve, J.-P. Katoen, V.Y. Nguyen, B. Postma and Y. Yushtein. 'Formal Correctness, Safety, Dependability and Performance Analysis of a Satellite'. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012 (cit. on p. 7).

[Fes+09] L.M. Fesq, G. Cancro, C. Jones, M. Ingham, J. Leitner, J. McDougal, M. Newhouse, E. Rice, D. Watson and J. Wertz. *Spacecraft Fault Management Workshop Results*. White Paper Report. NASA, Mar. 2009 (cit. on pp. 23–25, 27).

[FG88] B. Fox and P. Glynn. 'Computing Poisson Probabilities'. In: *Communications of the ACM* 31.4 (1988), pp. 440–445 (cit. on p. 137).

[FQ03] C. Flanagan and S. Qadeer. 'Thread-Modular Model Checking'. In: *Proceedings of the 10th International Conference on Model Checking Software (SPIN)*. Ed. by T. Ball and S.K. Rajamani. Vol. 2648. Lecture Notes in Computer Science. Springer, 2003, pp. 213–224 (cit. on p. 132).

[Gam+95] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1995 (cit. on p. 64).

[GPB02] D. Giannakopoulou, C.S. Pasareanu and H. Barringer. 'Assumption Generation for Software Component Verification'. In: *Proceedings of the 17th International Conference on Automated Software Engineering (ASE)*. IEEE, 2002, pp. 3–12 (cit. on p. 133).

[GPR11a] A. Gupta, C. Popeea and A. Rybalchenko. 'Predicate abstraction and refinement for verifying multi-threaded programs'. In: *Proceedings of the 38th Symposium on Principles of Programming Languages (POPL)*. Ed. by T. Ball and M. Sagiv. ACM, 2011, pp. 331–344 (cit. on p. 133).

[GPR11b] A. Gupta, C. Popeea and A. Rybalchenko. 'Threader: A Constraint-Based Verifier for Multi-Threaded Programs'. In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 412–417 (cit. on p. 132).

[Gru08] L. Grunske. 'Specification Patterns for Probabilistic Quality Properties'. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. Ed. by W. Schäfer, M.B. Dwyer and V. Gruhn. ACM, 2008, pp. 31–40 (cit. on pp. 64, 107).

[GS00] S. Garren and R. Smith. 'Estimating the Second Largest Eigenvalue of a Markov Transition Matrix'. In: *Bernoulli* 6.2 (2000), pp. 215–242 (cit. on p. 140).

[Guc+12] D. Guck, T. Han, J.-P. Katoen and M.R. Neuhäußer. 'Quantitative Timed Analysis of Interactive Markov Chains'. In: *Proceedings of the 4th NASA Formal Methods Symposium (NFM)*. Ed. by A.E. Goodloe and S. Person. Vol. 7226. Lecture Notes in Computer Science. Springer, 2012, pp. 8–23 (cit. on pp. 77, 90).

[Hec07] M. Hecht. *Software Reliability: Requirements Issues in Large Software Intensive Systems*. Presentation at Systems and Software Technology Conference 2007. June 2007 (cit. on p. 20).

[Hel07] L. Helander. 'Implementation and Evaluation of an Interpolation Based Model Checker'. Master's thesis. KTH Royal Institute of Technology, 2007 (cit. on pp. 120, 123).

[Her02] H. Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*. Vol. 2428. Lecture Notes in Computer Science. Springer, 2002 (cit. on pp. 74, 88, 113).

[HJ86] R. Horn and C. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, 1986 (cit. on pp. 137, 140).

[HL97] M. Hochbruck and C. Lubich. 'On Krylov Subspace Approximations to the Matrix Exponential Operator'. In: *Journal on Numerical Analysis* 34.5 (1997), pp. 1911–1925 (cit. on pp. 138, 140, 141, 151).

[HMS99] H. Hermanns, J. Meyer-Kayser and M. Siegle. 'Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains'. In: *Proceedings of the 3rd International Workshop on Numerical Solution of Markov Chains (NSMC)*. Ed. by B. Plateau, W. Stewart and M. Silva. Prensas Universitarias de Zaragoza, 1999, pp. 188–207 (cit. on p. 143).

[HMW09] T. Henzinger, M. Mateescu and V. Wolf. 'Sliding Window Abstraction for Infinite Markov Chains'. In: *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*. Ed. by A. Bouajjani and O. Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 337–352 (cit. on p. 149).

[HS11] F. Herbreteau and B. Srivathsan. ‘Coarse Abstractions Make Zeno Behaviours Difficult to Detect’. In: *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR)*. Ed. by J.-P. Katoen and B. König. Vol. 6901. Lecture Notes in Computer Science. Springer, 2011, pp. 92–107 (cit. on p. 69).

[HSW12] F. Herbreteau, B. Srivathsan and I. Walukiewicz. ‘Better Abstractions for Timed Automata’. In: *Proceedings of the 27th Symposium on Logic in Computer Science (LICS)*. 2012 (cit. on p. 69).

[IE11] R. Inc. and EUROCAE. *DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification*. Software Standard. 2011 (cit. on p. 114).

[IT90] O. Ibe and K. Trivedi. ‘Stochastic Petri-Net Models of Polling Systems’. In: *Selected Areas in Communications* 8.9 (1990), pp. 1649–1657 (cit. on p. 143).

[Jen53] A. Jensen. ‘Markoff Chains as an Aid in the Study of Markoff Processes’. In: *Scandinavian Actuarial Journal* 36 (1953), pp. 87–91 (cit. on p. 136).

[JM05] R. Jhala and K.L. McMillan. ‘Interpolant-Based Transition Relation Approximation’. In: *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*. Ed. by K. Etessami and S.K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 39–51 (cit. on p. 134).

[JVB07] A. Joshi, S. Vestal and P. Binns. ‘Automatic Generation of Static Fault Trees from AADL Models’. In: *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2007 (cit. on p. 73).

[Kat+11] J.-P. Katoen, I.S. Zapreev, E.M. Hahn, H. Hermanns and D.N. Jansen. ‘The Ins and Outs of the Probabilistic Model Checker MRMC’. In: *Performance Evaluation* 68.2 (2011), pp. 90–104 (cit. on pp. 95, 142, 143).

[KNP06] M. Kwiatkowska, G. Norman and D. Parker. ‘Symmetry Reduction for Probabilistic Model Checking’. In: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*. Ed. by T. Ball and R.B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 234–248 (cit. on p. 143).

[Kon+03] S. Konrad, L.A. Campbell, B.H.C. Cheng and M. Deng. ‘A Requirements Patterns-Driven Approach to Specify Systems and Check Properties’. In: *Proceedings of the 10th International Conference on Model Checking Software (SPIN)*. Ed. by T. Ball and S.K. Rajamani. Vol. 2648. Lecture Notes in Computer Science. Springer, 2003, pp. 18–33 (cit. on p. 64).

[KZ06] J.-P. Katoen and I. Zapreev. 'Safe On-The-Fly Steady-State Detection for Time-Bounded Reachability'. In: *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems (QEST)*. IEEE, 2006, pp. 301–310 (cit. on p. 147).

[Li+11] Y. Li, A. Zhu, C.-Y. Ma and M. Xu. 'A Method for Constructing Fault Trees from AADL Models'. In: *Proceedings of the 8th International Conference on Autonomic and Trusted Computing (ATC)*. Ed. by J.M.A. Calero, L.T. Yang, F.G. Mármol, L.J. García-Villalba, X.A. Li and Y. Wang. Vol. 6906. Lecture Notes in Computer Science. Springer, 2011, pp. 243–258 (cit. on p. 73).

[Lom+10] A. Lomuscio, B. Strulo, N. Walker and P. Wu. 'Assume-Guarantee Reasoning with Local Specifications'. In: *Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering (ICFEM)*. Ed. by J.S. Dong and H. Zhu. Vol. 6447. Lecture Notes in Computer Science. Springer, 2010, pp. 204–219 (cit. on p. 133).

[McM03] K. McMillan. 'Interpolation and SAT-Based Model Checking'. In: *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*. Ed. by W. Hunt and F. Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 1–13 (cit. on pp. 128, 134).

[McM05] K.L. McMillan. 'Applications of Craig Interpolants in Model Checking'. In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by N. Halbwachs and L.D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 1–12 (cit. on pp. 119, 127, 134).

[Mey92] J.F. Meyer. 'Performability: A Retrospective and Some Pointers to the Future'. In: *Performance Evaluation* 14.3–4 (1992), pp. 139 –156 (cit. on p. 87).

[ML03] C. Moler and C.V. Loan. 'Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later'. In: *SIAM Review* 45.1 (2003), pp. 3–49 (cit. on p. 135).

[MLK04] M. Massink, D. Latella and J.-P. Katoen. 'Model Checking Dependability Attributes of Wireless Group Communication'. In: *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2004, pp. 711–720 (cit. on p. 144).

[MPR06] A. Malkis, A. Podelski and A. Rybalchenko. 'Thread-modular Verification is Cartesian Abstract Interpretation'. In: *Proceedings of the 3th International Conference on Theoretical Aspects of Computing (ICTAC)*. Ed. by K. Barkaoui, A. Cavalcanti and A. Cerone. Vol. 4281. Lecture

Notes in Computer Science. Springer, 2006, pp. 183–197 (cit. on p. 132).

[MPR10] A. Malkis, A. Podelski and A. Rybalchenko. 'Thread-Modular Counterexample Guided Abstraction Refinement'. In: *Proceedings of the 17th International Conference on Static Analysis (SAS)*. Ed. by R. Cousot and M. Martel. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, pp. 356–372 (cit. on p. 133).

[MS94] A. van Moorsel and W. Sanders. 'Adaptive Uniformization'. In: *Communications in Statistics - Stochastic Models* 10.3 (1994), pp. 619–648 (cit. on p. 149).

[Nol11a] T. Noll. *Specification of SLIM Language Graphical Notation*. Issue 1.1 D1. RWTH Aachen University, May 2011 (cit. on p. 50).

[Nol11b] T. Noll. *Specification of the COMPASS System-Level Integrated Modeling (SLIM) Language*. Tech. rep. RWTH Aachen University, July 2011 (cit. on pp. 29, 93).

[NR08] V.Y. Nguyen and T.C. Ruys. 'Incremental Hashing for SPIN'. In: *Proceedings of the 15th International Conference on Model Checking Software (SPIN)*. Ed. by K. Havelund, R. Majumdar and J. Palsberg. Vol. 1556. Lecture Notes in Computer Science. Springer, 2008, pp. 232–249 (cit. on p. 7).

[NR09] V.Y. Nguyen and T.C. Ruys. 'Memoised Garbage Collection for Software Model Checking'. In: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by S. Kowalewski and A. Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 201–214 (cit. on p. 7).

[NR13] V.Y. Nguyen and T.C. Ruys. 'Selected Dynamic Issues in Software Model Checking'. In: *Software Tools for Technology Transfer (STTT)* 15.4 (2013), pp. 337–362 (cit. on p. 7).

[Ode10] M.R. Odenbrett. 'Explicit-State Model Checking of an Architectural Design Language using SPIN'. Diplomarbeit. RWTH Aachen University, 2010 (cit. on pp. 67, 95).

[ONN10] M.R. Odenbrett, V.Y. Nguyen and T. Noll. 'Slicing AADL Specifications for Model Checking'. In: *Proceedings of the 2nd NASA Formal Methods Symposium (NFM)*. Ed. by C. Muñoz. NASA Conference Proceedings, 2010, pp. 217–221 (cit. on pp. 5, 7).

[Par07] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007 (cit. on p. 93).

[Per+12] M. Perrotin, E. Conquet, J. Delange, A. Schiele and T. Tsiodras. 'TASTE: A Real-Time Software Engineering Tool-Chain Overview, Status, and Future'. In: *Proceedings of the 15th International SDL Forum (SDL)*. Ed. by I. Ober and I. Ober. Vol. 7083. Lecture Notes in Computer Science. Springer, 2012, pp. 26–37 (cit. on p. 54).

[Rin55] R.F. Rinehart. 'The Equivalence of Definitions of a Matric Function'. In: *The American Mathematical Monthly* 62.6 (1955), pp. 395–414 (cit. on pp. 135, 140).

[RKH09] X. Renault, F. Kordon and J. Hugues. 'Adapting Models to Model Checkers, A Case Study : Analysing AADL Using Time or Colored Petri Nets'. In: *Proceedings of the 20th International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2009, pp. 26–33 (cit. on p. 54).

[RKK07] A.-E. Rugina, K. Kanoun and M. Kaâniche. 'A System Dependability Modeling Framework Using AADL and GSPNs'. In: *Proceedings of the Workshop on Software Architectures for Dependable Systems (WADS)*. Ed. by R. de Lemos, C. Gacek and A.B. Romanovsky. Vol. 4615. Lecture Notes in Computer Science. Springer, 2007, pp. 14–38 (cit. on p. 55).

[RSS12] S.F. Rollini, O. Sery and N. Sharygina. 'Leveraging interpolant strength in model checking'. In: *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*. Ed. by P. Madhusudan and S.A. Seshia. Lecture Notes in Computer Science. Springer, 2012, pp. 193–209 (cit. on p. 128).

[Saa92] Y. Saad. 'Analysis of Some Krylov Subspace Approximations to the Matrix Exponential Operator'. In: *SIAM Journal on Numerical Analysis* 29.1 (1992), pp. 209–228 (cit. on pp. 137, 138, 140, 141).

[Sid98] R. Sidje. 'Expokit: a software package for computing matrix exponentials'. In: *Transactions on Mathematical Software (TOMS)* 24.1 (1998), pp. 130–156 (cit. on pp. 140, 151).

[Sin+05] F. Singhoff, J. Legrand, L. Nana and L. Marcé. 'Scheduling and Memory Requirements Analysis with AADL'. In: *Proceedings of the International Conference on ADA (SIGAda)*. Ed. by J.W. McCormick and L.C. Baird. ACM, 2005, pp. 1–10 (cit. on p. 54).

[Smi+02] R.L. Smith, G.S. Avrunin, L.A. Clarke and L.J. Osterweil. 'PROPEL: An Approach Supporting Property Elucidation'. In: *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. Ed. by W. Tracz, M. Young and J. Magee. ACM, 2002, pp. 11–21 (cit. on p. 64).

[SP07] A. Schumann and Y. Pencolé. 'Scalable diagnosability checking of event-driven systems'. In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI)*. Ed. by M.M. Veloso. Morgan Kaufmann, 2007, pp. 575–580 (cit. on p. 84).

[SSS96] R. Sidje, R. Sidje and W.J. Stewart. 'A Survey of Methods for Computing Large Sparse Matrix Exponentials Arising in Markov Chains'. In: *Computational Statistics and Data Analysis* 29 (1996), pp. 345–368 (cit. on p. 135).

[TBI97] L. Trefethen and D. Bau III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997 (cit. on p. 140).

[Tri02] S. Tripakis. 'Fault Diagnosis for Timed Automata'. In: *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*. Ed. by W. Damm and E.-R. Olderog. Vol. 2469. Lecture Notes in Computer Science. Springer, 2002, pp. 205–224 (cit. on p. 84).

[Tri99] S. Tripakis. 'Verifying Progress in Timed Systems'. In: *Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*. Ed. by J.-P. Katoen. Vol. 1601. Lecture Notes in Computer Science. Springer, 1999, pp. 299–314 (cit. on pp. 68, 69).

[War77] R. Ward. 'Numerical Computation of the Matrix Exponential With Accuracy Estimate'. In: *SIAM Journal on Numerical Analysis* 14.4 (1977), pp. 600–610 (cit. on pp. 139, 140).

[Weg87] I. Wegener. *The Complexity of Boolean Functions*. Wiley-Teubner, 1987 (cit. on p. 71).

[Wei94] R. Weiss. 'Error-Minimizing Krylov Subspace Methods'. In: *SIAM Journal on Scientific Computing* 15.3 (1994), pp. 511–527 (cit. on p. 151).

[WHM09] C.M. Wintersteiger, Y. Hamadi and L. Moura. 'A Concurrent Portfolio Approach to SMT Solving'. In: *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*. Ed. by A. Bouajjani and O. Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 715–720 (cit. on p. 116).

[Wim+06] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp and B. Becker. 'Sigref - A Symbolic Bisimulation Tool Box'. In: *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Ed. by S. Graf and W. Zhang. Vol. 4218. Lecture Notes in Computer Science. Springer, 2006, pp. 477–492 (cit. on pp. 94, 110).

[Yus+11] Y. Yushtein, M. Bozzano, A. Cimatti, J. Katoen, V.Y. Nguyen, T. Noll, X. Olive and M. Roveri. ‘System-Software Co-Engineering: Dependability and Safety Perspective’. In: *Proceedings of the 4th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. IEEE, 2011, pp. 18–25 (cit. on p. 7).

[Zap08] I. Zapreev. ‘Model Checking Markov Chains: Techniques and Tools’. PhD Thesis. University of Twente, 2008 (cit. on p. 147).

[ZN10] L. Zhang and M.R. Neuhäußer. ‘Model Checking Interactive Markov Chains’. In: *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by J. Esparza and R. Majumdar. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 53–68 (cit. on pp. 76, 90).

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2010-01 *	Fachgruppe Informatik: Jahresbericht 2010
2010-02	Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03	Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04	René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-05	Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
2010-06	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
2010-07	George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08	Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09	George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs
2010-10	Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut
2010-11	Martin Zimmermann: Parametric LTL Games
2010-12	Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut
2010-13	Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems
2010-14	Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination
2010-15	Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode
2010-16	Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

2010-17	Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten
2010-18	Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit
2010-19	Felix Reidl: Experimental Evaluation of an Independent Set Algorithm
2010-20	Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games
2011-01 *	Fachgruppe Informatik: Jahresbericht 2011
2011-02	Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
2011-03	Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems
2011-04	Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars
2011-06	Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++
2011-07	Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV
2011-08	Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
2011-09	Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
2011-10	Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-11	Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains
2011-12	Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13	Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14	Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16	Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-17	Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis
2011-18	Kamal Barakat: Introducing Timers to pi-Calculus
2011-19	Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode
2011-24	Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations

2011-25	Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26	Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01	Fachgruppe Informatik: Annual Report 2012
2012-02	Thomas Heer: Controlling Development Processes
2012-03	Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04	Marcus Gelderie: Strategy Machines and their Complexity
2012-05	Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting
2012-06	Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data
2012-07	André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms
2012-08	Hongfei Fu: Computing Game Metrics on Markov Decision Processes
2012-09	Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains
2012-10	Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations
2012-12	Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs
2012-15	Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations
2012-16	Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets
2013-01 *	Fachgruppe Informatik: Annual Report 2013
2013-02	Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen
2013-03	Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM
2013-04	Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries
2013-05	Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013
2013-06	Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation

2013-08	Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers
2013-10	Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata

[*] These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.