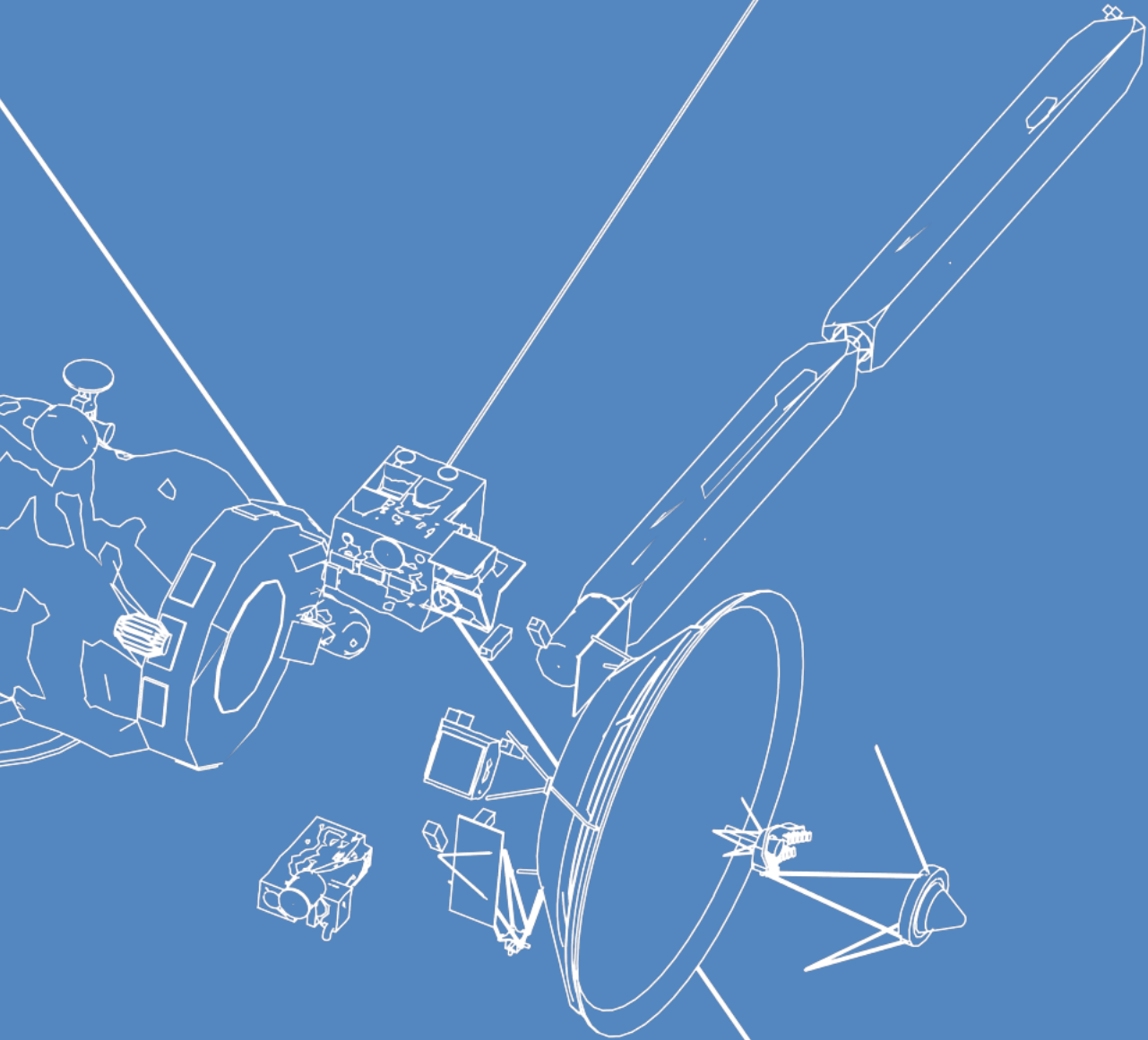


# Model-Based Reliability Analysis of Aerospace Systems



Harold Y. Bruintjes



# **Model-Based Reliability Analysis of Aerospace Systems**

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

vorgelegt von

**Harold Y. Brountjes, Master of Science**

aus

**Wijk bij Duurstede, die Niederlande**

Berichter: Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen  
P.D. Alessandro Cimatti, MSc.

Tag der mündlichen Prüfung: 2018-09-27

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.



# ABSTRACT

In order to provide reliable and safe systems in the aerospace domain, despite increased complexity and stronger demands on capabilities, new model-based system and software engineering approaches are being developed. One of which is COMPASS. COMPASS offers an integrated approach to the analysis of correctness, safety and performability of (aerospace) systems, providing a multitude of functions using a single input specification language.

This thesis starts with describing the modeling language that is employed by the COMPASS toolset, both its syntax and semantics. Named SLIM, and derived from the AADL, it allows for the specification of nominal and error models, as well as their integration. Various modifications have been made since its first release, which are discussed in-depth.

The specification of requirements is treated next, in particular using formal representations. A taxonomy is defined that allows requirements to be classified. Subsequently, different approaches to formalizing requirements are treated, in particular pattern-based and design-attribute based approaches. The latter, combined with the taxonomy, leads to the definition of the catalogue of system and software properties, which encodes requirement specifications directly in the system model.

This is followed by a description of statistical model checking techniques. Such techniques allow for statistically determining, by means of simulation, outcomes for the model checking problem. In the setting of COMPASS, semantic issues may arise, which are classified and addressed. A tool making use of this knowledge is described, which is used as a probabilistic engine in the COMPASS toolset.

As over the past years the COMPASS toolset has seen numerous enhancements, leading to its latest release— version 3.0. Therefore, its capabilities and architecture are revisited as well. New additions include the support for timed failure propagation graphs, non-deterministic models in probabilistic analysis, contract-based analysis, and the newly developed statistical model checking and requirement formalization approaches.

Finally, to make the toolset more accessible, both a demonstrator and a case study have been constructed. The demonstrator deploys a SLIM-based specification on a LEGO® satellite model, making use of a Simulink translator to facilitate this. The case study is based on one performed for the CubETH project, describing part of its software architecture.



# ZUSAMMENFASSUNG

Zur Sicherstellung von zuverlässigen und sicheren Systemen, trotz steigender Komplexität und hoher Nachfrage nach Leistungsfähigkeit, werden neuartige Ansätze zur modellbasierten System- und Anwendungsentwicklung im Bereich der Luft- und Raumfahrt erarbeitet. Einer dieser Ansätze ist COMPASS. COMPASS bietet einen integrativen Ansatz zur Analyse von Korrektheit, Sicherheit und Leistungsfähigkeit im Luft- und Raumfahrtbereich und stellt eine Vielzahl von Funktionen mit Hilfe einer einzelnen Spezifikationsprache zur Verfügung.

Die vorliegende Arbeit beginnt mit einer Beschreibung der verwendeten Modellierungssprache. Die Sprache SLIM, welche sich aus der Sprache AADL ableitet, erlaubt die Spezifikation und Integration von Nominal- und Fehlermodellen. Verschiedenste Abwandlungen werden im Detail erläutert.

Im Anschluss hieran wird die Anforderungsspezifikation behandelt, wobei besonderes Augenmerk auf formale Darstellungen gelegt wird. Eine Taxonomie wird definiert, welche es erlaubt unterschiedliche Arten von Anforderungen zu klassifizieren. Anschließend werden verschiedene Ansätze zur Formalisierung von Anforderungen, speziell musterbasierte und auf Entwurfseigenschaften basierende, entwickelt. Letztere führen, zusammen mit der Taxonomie, zur Definition der CSSP, welche Anforderungsspezifikationen direkt im Systemmodell kodiert.

Diesem folgt eine Beschreibung der Techniken zur statistischen Modellüberprüfung, welche es erlauben statisch — durch Simulation — das Ergebnis der Modellüberprüfung festzustellen. Im Kontext von COMPASS auftretende, semantische Probleme werden klassifiziert und Lösungswege aufgezeigt. Basierend hierauf wird ein Programm vorgestellt, welches dieses Wissen ausnutzt und von einer probabilistischen Einheit in COMPASS verwendet wird.

Neben einer Vielzahl von Verbesserungen in COMPASS wurden auch seine Möglichkeiten und Architektur überholt. Zum Zeitpunkt dieser Arbeit stellt Version 3.0 den aktuellen Stand dar. Erweiterungen hierin beinhalten die Unterstützung von „Timed Failure Propagation Graphs“, nicht-deterministischen Modellen in probabilistischen Analysen, „Contract-Based Analysis“, sowie die neuentwickelte statistische Modellüberprüfung und Ansätze zur Formalisierung von Anforderungen.

Schlussendlich wurden sowohl ein Demonstrator als eine Fallstudie erstellt, um COMPASS zugänglicher zu machen. Der Demonstrator verwendet hierbei eine auf SLIM basierende Spezifikation eines Satellitenmodells. Um dies zu erleichtern wird hierfür eine Übersetzung eines Simulink-Modells benutzt. Die vorgestellte Fallstudie basiert auf einer im Rahmen des Projekts durchgeführten Fallstudie und beschreibt Teile der Softwarearchitektur.



# FOREWORD

This thesis would not have come to completion had it not been for the support of my advisors, colleagues, friends and family. In contrast to how the topics in this thesis may concern how things can go wrong, they ensured many things went right, and for that I would like to express my appreciation.

When I was offered the opportunity to work on the COMPASS project at the chair of Joost-Pieter, I gladly took it. It resulted in a rather enjoyable time at the chair, and interesting topics to work on. And so I would like to thank my supervisor Joost-Pieter Katoen, for providing me the organization, coordination, guidance and valuable insights that were needed to complete this work. Thanks also go to Thomas Noll, both for assisting me and his additional work for keeping the COMPASS project on track. Without Joost-Pieters and Thomas' help, the project would not have run nearly as smoothly, or be as interesting, as it did.

I started working on the COMPASS project together with Viet Yen Nguyen, already a veteran, and our colleagues at FBK. Not much later I was assisted by Dimitri Bohlender, Sebastian Junges and Jens Katelaan, who proved to be excellent minions. I would like to thank them for all their help during the various follow-up projects, and in particular Viet Yen for helping me getting acquainted with the life of a PhD student. Thanks also to everyone at FBK, in particular Alessandro, Marco and Stefano, it was a pleasure working together with you on the project. Further thanks go to Alessandro for being my second supervisor as well.

The chair itself provided a great environment to work in, with people that worked amazingly well together, sharing ideas and helping each other along. Thanks Tim, Philipp and Sebastian for being the best possible office mates. The countless number of laughs we had together were very enjoyable, though, allegedly, not everyone shared that sentiment. Also kudos to all my peers at the chair, for all their support, ideas and fun we had, but maybe more importantly for not unleashing their wrath when I inevitably broke some of our IT again. That was a great learning experience, and I hope I hid the remaining bodies well enough.

I would also like to thank Elke and Birgit for taking care of the cliché that is German bureaucracy, as well as organizing a lot of the more pleasant events that occurred at and around the chair, and inventing all the ways to keep that well within regulation.

Thanks also go to my family and friends, as there is more to this than the people at the office. My parents, for their invariable moral support and providing that home away from home (it was worth buying a car just for that alone). Tom, for showing me the way, both on paper and some distance above sea-level. Also, my apologies for making your predictions off-by-one. Then again, phase arrays can

hardly be reliably predicted, can they? Maudy, for helping me design the cover, perhaps answering the hardest question of all. Freark, Djurre, Bart and Robbert, it is always fun when I come visiting and vice versa, and likewise for our gaming matches.

# CONTENTS

vii

CONTENTS

<b>ABSTRACT</b>	<b>i</b>
<b>ZUSAMMENFASSUNG</b>	<b>iii</b>
<b>FOREWORD</b>	<b>v</b>
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 VERIFICATION OF (AEROSPACE) SYSTEMS . . . . .	2
1.2 APPROACH . . . . .	3
1.2.1 PROBLEM STATEMENT . . . . .	3
1.3 OUTLINE . . . . .	4
1.4 PRIOR PUBLICATIONS . . . . .	5
1.5 CONTRIBUTIONS . . . . .	6
<b>2 MODELING</b>	<b>9</b>
2.1 SLIM AND MODELING AEROSPACE SYSTEMS . . . . .	10
2.2 MODELING SYSTEMS . . . . .	10
2.2.1 COMPONENT TYPES . . . . .	11
2.2.2 COMPONENT IMPLEMENTATIONS . . . . .	12
2.2.3 COMPOSITION AND RECONFIGURATION . . . . .	14
2.2.4 PACKAGES . . . . .	17
2.2.5 DATA . . . . .	18
2.2.6 CONSTANTS . . . . .	19
2.2.7 HYBRID MODELS . . . . .	19
2.2.8 PROPERTIES . . . . .	20
2.3 ERROR MODELING . . . . .	22
2.4 MODEL EXTENSION . . . . .	24
2.4.1 FAULT INJECTIONS . . . . .	24
2.4.2 ERROR PROPAGATIONS . . . . .	25
2.4.3 MODEL TRANSFORMATION . . . . .	26

2.5	SEMANTICS . . . . .	30
2.5.1	EDAs . . . . .	30
2.5.2	NEDAs . . . . .	34
2.5.3	NON-DETERMINISM . . . . .	39
2.5.4	PROBABILISTIC SEMANTICS . . . . .	39
2.5.5	INVALID BEHAVIORS . . . . .	41
2.6	DIFFERENCES BETWEEN SLIM AND AADL . . . . .	41
2.7	FUTURE WORK . . . . .	43
2.7.1	AADL v3 . . . . .	43
<b>3</b>	<b>SPECIFYING REQUIREMENTS</b> . . . . .	<b>45</b>
3.1	REQUIREMENTS IN AEROSPACE . . . . .	46
3.1.1	ABSTRACTION LEVELS . . . . .	46
3.2	CLASSIFYING REQUIREMENTS . . . . .	47
3.3	FORMAL ANALYSIS . . . . .	51
3.3.1	FORMALIZATION . . . . .	52
3.3.2	MODEL CHECKING . . . . .	57
3.3.3	REQUIREMENTS REFINEMENT AND CONTRACT BASED DESIGN . . . . .	57
3.4	CATALOGUE OF SYSTEM AND SOFTWARE PROPERTIES . . . . .	59
3.4.1	PROPERTIES AS DESIGN ATTRIBUTES . . . . .	59
3.4.2	CSSP PROPERTY SET . . . . .	60
3.4.3	CSSP DERIVED PROPERTIES . . . . .	60
3.5	DISCUSSION . . . . .	61
<b>4</b>	<b>STATISTICAL MODEL CHECKING</b> . . . . .	<b>63</b>
4.1	ACCURACY AND PRECISION . . . . .	64
4.2	STATISTICAL ANALYSIS . . . . .	65
4.2.1	HYPOTHESIS TESTING . . . . .	66
4.2.2	PROBABILITY ESTIMATION . . . . .	68
4.2.3	RELATED WORK AND TOOLS . . . . .	68
4.3	CAVEATS IN SIMULATION SEMANTICS . . . . .	69
4.3.1	TRANSITION INTERLEAVING . . . . .	70
4.3.2	UNDERSPECIFICATION . . . . .	70
4.3.3	SCHEDULING . . . . .	72
4.3.4	SYNCHRONIZATION . . . . .	72
4.3.5	BIAS . . . . .	73
4.3.6	STRICT BOUNDS . . . . .	73
4.3.7	INVALID PATHS . . . . .	74
4.3.8	PARALLELIZATION . . . . .	75
4.3.9	RARE EVENTS . . . . .	75

4.4	CLASSIFICATION OF IMPLEMENTATIONS . . . . .	76
4.4.1	ORDER . . . . .	76
4.4.2	ACCURACY . . . . .	76
4.4.3	SCOPE . . . . .	77
4.4.4	RACE POLICY . . . . .	77
4.5	STRATEGIES . . . . .	78
4.6	SLIMSIM . . . . .	79
4.6.1	INPUT . . . . .	79
4.6.2	MODEL REALIZATION . . . . .	80
4.6.3	SIMULATION . . . . .	81
4.6.4	DESIGN CHOICES . . . . .	82
4.7	CASE STUDY . . . . .	83
4.7.1	NOMINAL MODEL . . . . .	85
4.7.2	ERROR MODEL . . . . .	86
4.7.3	EXPERIMENTAL RESULTS . . . . .	86
4.8	COMPARISON WITH PERFORMABILITY . . . . .	88
4.9	DISCUSSION . . . . .	89
<b>5</b>	<b>THE COMPASS TOOLSET . . . . .</b>	<b>91</b>
5.1	(SUB)PROJECTS . . . . .	92
5.2	ARCHITECTURE . . . . .	94
5.2.1	MODELS . . . . .	94
5.2.2	PROPERTIES . . . . .	96
5.2.3	MODEL TRANSLATION . . . . .	96
5.2.4	BACK-END TOOLS . . . . .	99
5.2.5	USER INTERFACE . . . . .	101
5.3	FUNCTIONS . . . . .	102
5.3.1	PROPERTY SPECIFICATION . . . . .	102
5.3.2	MISSION SPECIFICATION . . . . .	106
5.3.3	CORRECTNESS ANALYSIS . . . . .	106
5.3.4	CONTRACT BASED ANALYSIS . . . . .	109
5.3.5	PERFORMABILITY . . . . .	111
5.3.6	SAFETY AND DEPENDABILITY . . . . .	113
5.3.7	FAULT DETECTION, ISOLATION AND RECOVERY . . . . .	117
5.3.8	DIAGNOSABILITY ANALYSIS . . . . .	118
5.3.9	TIMED FAILURE PROPAGATION GRAPHS . . . . .	119
5.4	FUTURE WORK/ROADMAP . . . . .	122
5.5	DISCUSSION . . . . .	124

<b>6</b>	<b>COMPASS DEMONSTRATOR</b>	<b>125</b>
6.1	SIMULINK	126
6.1.1	STATEFLOW	126
6.1.2	SIMULATION AND EXECUTION	127
6.2	SIMULINK TRANSLATOR	128
6.2.1	COMPONENTS	128
6.2.2	CONNECTIONS AND FLOWS	129
6.2.3	MODES AND STATES	130
6.2.4	TYPES AND EXPRESSIONS	131
6.2.5	RECONFIGURATIONS	133
6.2.6	SIMULINK AND CYCLES	133
6.2.7	ERROR MODELS	134
6.2.8	GLOBAL EVENT SEMAPHORE	134
6.2.9	LIMITATIONS	135
6.3	DEMONSTRATOR	135
6.3.1	SLIM MODEL	136
6.3.2	LEGO® MODEL	138
6.3.3	DEPLOYMENT	139
6.4	DISCUSSION	140
6.4.1	FUTURE WORK	141
<b>7</b>	<b>MODELING AND ANALYSIS OF THE CUBETH NANO-SATELLITE IN SLIM</b>	<b>143</b>
7.1	CUBETH PROJECT	144
7.2	BIP	145
7.2.1	FROM BIP MODELS TO SLIM MODELS	146
7.3	ARCHITECTURE PATTERNS	147
7.3.1	MUTUAL EXCLUSION	150
7.3.2	CLIENT-SERVER	151
7.3.3	ACTION FLOW	152
7.3.4	FAILURE MONITORING	154
7.3.5	MODE MANAGEMENT	155
7.3.6	BUFFER MANAGEMENT	156
7.3.7	EVENT MONITORING	156
7.3.8	PRIORITY MANAGEMENT	157
7.4	MODEL	158
7.4.1	REQUIREMENTS	158
7.4.2	COMPONENTS	162
7.4.3	DIFFERENCES BETWEEN SLIM AND BIP	175
7.4.4	FAIRNESS	176
7.5	ANALYSIS	176

7.5.1	PROPERTIES . . . . .	177
7.5.2	VERIFICATION . . . . .	181
7.6	DISCUSSION . . . . .	182
<b>8</b>	<b>CONCLUSION</b> . . . . .	<b>185</b>
8.1	CHALLENGES AND FUTURE WORK . . . . .	186
<b>A</b>	<b>LOGICS</b> . . . . .	<b>189</b>
A.1	LINEAR TEMPORAL LOGIC . . . . .	190
A.2	COMPUTATION TREE LOGIC . . . . .	190
A.3	PROBABILISTIC CTL . . . . .	191
A.4	CONTINUOUS STOCHASTIC LOGIC . . . . .	191
A.5	METRIC TEMPORAL LOGIC . . . . .	192
A.6	SLIM LOGICAL EXPRESSIONS . . . . .	192
A.7	OPERATOR PRECEDENCE . . . . .	194
<b>B</b>	<b>PROPERTY PATTERNS</b> . . . . .	<b>195</b>
B.1	GRAMMAR . . . . .	195
B.2	FORMULAS . . . . .	196
B.2.1	EXISTENCE . . . . .	196
B.2.2	UNIVERSALITY . . . . .	197
B.2.3	ABSENSE . . . . .	198
B.2.4	RECURRENCE . . . . .	198
B.2.5	PRECEDENCE . . . . .	199
B.2.6	RESPONSE . . . . .	200
B.2.7	RESPONSE-INVARIANCE . . . . .	200
B.2.8	UNTIL . . . . .	201
<b>C</b>	<b>CSSP</b> . . . . .	<b>203</b>
C.1	CSSP PROPERTY SET . . . . .	203
C.2	CSSP FORMAL PROPERTIES . . . . .	206
C.3	CSSP FORMAL DEFINITIONS . . . . .	208
C.4	DESIGN ATTRIBUTE MAPPING . . . . .	209

<b>D</b>	<b>OUTPUT/ARTIFACTS</b>	<b>213</b>
D.1	TRACES . . . . .	213
D.2	FAULT TREES . . . . .	213
D.2.1	DYNAMIC FAULT TREES . . . . .	214
D.3	TIMED FAILURE PROPAGATION GRAPHS . . . . .	216
	<b>ACRONYMS</b>	<b>219</b>
	<b>GLOSSARY</b>	<b>225</b>
	<b>BIBLIOGRAPHY</b>	<b>229</b>
	<b>STANDARDS</b>	<b>243</b>
	<b>PRIOR PUBLICATIONS</b>	<b>245</b>



# 1

## INTRODUCTION

**M**OVING ever further, Voyager I [138] became the first man-made space probe to leave our solar system on August 25, 2012, nearly 35 years after its launch on September 5, 1977. For this achievement to be possible, the Voyager's design had to be absolutely *reliable*. Although its primary mission had ended already at the end of 1980, it is expected to last until 2025 when its generators can no longer supply enough power for the space probe to perform its mission — a total mission time of close to 50 years.

This result raises some interesting questions: What is it that allows systems to remain operational for such a long time, and how can we make sure that this also holds for future projects? Unfortunately, simply repeating what was done before does not suffice. One major challenge in answering these questions is the ever growing size and complexity of systems all around us, and the ever strong dependence on software. What worked well in the past may no longer be applicable in the present — Voyager I uses processors running at less than 1 MHz and stores its data on tape [139], technology that has long since been obsoleted and superseded.

As the number of components of a system grows, and the complexity of any software running on it rises, the likelihood of a failure occurring within a given timespan increases. After all, each extra component adds one more that can fail. For example, electronics in space are subject to a considerable amount of (background) radiation. With an ever-increasing density of semiconductors, the likelihood of radiation corrupting data or damaging components increases proportionally, requiring additional radiation hardening. Effectively, to mitigate the increased likelihood of failures that is a result of increasingly larger systems, extra measures have to be put in place to be able to deal with the occurrence of failures. Added redundancy in time or space (i.e., repeated information or duplicated functionality) allows for subsystems to fail, while the system itself remains operational (though perhaps in a degraded state).

Both the increase in complexity of systems, as well as the increasingly complex fault management approaches, introduce another problem: The effects of various events in the system become harder to understand, making it more likely the system does not behave according to expectations — a particular tough problem when such behaviors occur only as a result of relatively rare events. One way to deal with this is to break the system into smaller parts, and verifying those to work correctly. The complete system can then be constructed from these parts. Alternatively, subsystems used successfully in previous configurations can be reused. Their interaction, however, must be well understood, as the assumptions made by the various integrated components have to be upheld by others. The effects of wrongfully making such assumptions are famously shown by the catastrophic failure of Ariane 5 Flight 501 shortly after launch.

Both issues described above stem from *complexity*, which impedes the ability of engineers to fully understand the behavior of the system under development. In order to make the design and implementation of such systems manageable, *rigor* is required along every step of the way, ensuring all decisions are well-founded and documented.

## 1.1 VERIFICATION OF (AEROSPACE) SYSTEMS

The aerospace domain, driven by a high demand for reliability, defines a large number of processes to ensure that any choices are made after careful consideration, validating the approach taken and verifying the outcome is correct. Many of them, for example, are defined by the standards set forth by the European Cooperation for Space Standardization (ECSS) [56].

However, most of these processes are subject to issues that stem from the problems mentioned previously:

- » Many of the processes are done manually. This means that they require a significant amount of expertise, are costly and time demanding. In addition, the risk of an error being made due to an oversight is much greater compared to an automated process.
- » There is not always a formal relation between the different artifacts that are produced as a result of the different processes during development. For example, traceability matrices between requirements and the system and software design are often made and maintained by hand. A change in the requirements or design is not automatically propagated, and care has to be taken that both they and the matrices remain consistent.
- » Verification is mostly performed by means of review, inspection or testing. Although these methods can be used to detect the presence of faults or inspire overall confidence, no hard guarantees can be made about the absence of faults.

The strong dependence on manual effort and lack of (automated) proofs make it more likely that a fault slips into the specification or design and remains undetected.

A method to deal with this, is a model-based system and software engineering (MBSSE) approach, or more generally model-driven engineering (MDE). The engineering of the system and software is based on a single model specified at the start of the project, and which continuously gets refined in later stages. The output generated as part of the various subsequent phases can be based on or derived from this model. At the least, such a model allows for a clear traceability between it and the derived artifacts, allowing for easy detection of missing or inconsistent information. Moving further, when such artifacts are generated based on well-defined mappings, this process can be automated, up to the point where a change in the model can automatically be propagated to the various derived elements.

Various initiatives have been started by the European Space Agency (ESA) that work toward employing a MBSSE approach for the development of future projects [51]. Among these are, e.g., the On-board Software Reference Architecture (OSRA), developed as parts of the SAVOIR [123] project, the TASTE project [133] and the COMPASS project [50]. It is the latter that defines the scope for the work described in this thesis. The COMPASS project had the goal of providing a “*theoretical and technological basis for the system-software co-engineering approach focusing on a coherent set of specification and analysis techniques for evaluation of system-level correctness, safety, dependability and performance (performability) of the on-board computer-based systems* [59].” Effectively, this was achieved by providing a single model specification language with a well-defined formal semantics, dubbed SLIM, that allows the various desired analyses to be performed. The latter has been made possible with the use of various back-end tools, such as model checkers.

## 1.2 APPROACH

COMPASS provides a general framework that can tackle the aforementioned problems. Since there is a single input language, all functions offered by the toolset can be used based on a single model specified using that language. This means that COMPASS has the potential to cover many of the aspects that are part of the early validation and verification activities during a project. The initial COMPASS project covered many of these aspects, and was later extended in follow-up projects to cover further needs. The work described in this thesis was performed as part of some of these projects, targeting specific needs for space projects (though in general applicable to others).

### 1.2.1 PROBLEM STATEMENT

One of the key strengths of the COMPASS toolset is its formal nature, combined with an interface that allows to abstract away from this for the user. The formal semantics of the SLIM language and analyses based on formal verification allow it to perform each function with mathematical rigor, making it possible to *prove*

that facts about the system, in particular with regard to correctness, safety and performability.

One problem, however, is on the specification side: Directly translating a specification conveyed in natural language to some formalism is a non-trivial task, fraught with its own risks. Problems may occur either because the specification has errors, or mistakes are made during the translation thereof. Ideally, such problems are avoided by either providing automation, or otherwise assistance, during translation, or simply constructing the specification in a way that is already backed by formal semantics. The first part of this thesis covers this area, discussing the SLIM language of COMPASS and ways of dealing with the specification and analysis of requirements.

On the other end it may be the case that the analysis of certain problems can be *hard*, even undecidable. This means that a straightforward approach does not always work, or that the analysis is not feasible with state-of-the-art implementations and hardware. Aside from searching for solutions for open problems, the answer lies in finding alternative approaches, or tailoring the solution to a specific subproblem that can effectively be tackled. This thesis covers this aspect in the second part, describing how statistical means can be used to provide probabilistic metrics, and how the COMPASS toolset itself implements its various functions.

### 1.3 OUTLINE

The complete contents of the following chapters are briefly described below:

- » Chapter 2 introduces the language used by the COMPASS toolset, SLIM. This is the language used to specify the input model used for the various functions of the COMPASS toolset. Both the syntax of the language and the semantics of the language are described. In addition, the model extension concept is explained, used to introduce error behavior into the model.
- » In Chapter 3, the specification and analysis of requirements is treated. A taxonomy of possible requirement types is given, as well as an overview of possible approaches to formalize requirements. In particular, formalization based on predefined patterns and formalization based on design attributes is treated.

The following two chapters then deal with the analysis thereof, in particular with regards to tooling:

- » In Chapter 4, the notion and use of statistical model checking (SMC) is introduced, which permits the probabilistic analysis of systems for which other methods may be intractable or no other analysis methods even exist. This chapter explains the concept behind statistical model checking, some of its caveats in the setting of this thesis and an implementation that is used within the COMPASS toolset.

- » Chapter 5 describes the COMPASS toolset. It covers the architecture of the toolset, its features and the artifacts that it produces. The chapter is based on the latest iteration at time of writing, which is version 3.0.

Finally, Chapters 6 and 7 deal with applications of the COMPASS toolset:

- » In Chapter 6 a demonstrator is described which is intended to showcase how the COMPASS toolset works in a more tangible manner. It describes a translation from SLIM to Simulink, which is used to deploy executable code on the LEGO® Mindstorms platform, which in turn is used to drive a LEGO® model of a satellite.
- » Chapter 7 describes the modeling and analysis of the command and data management system (CDMS) system of the CubETH nano-satellite, originally provided as a case study performed using the BIP framework. It has been translated to the COMPASS setting, making use of the SLIM language. The architecture patterns that were defined and used in the prior case study are treated here as well.

## 1.4 PRIOR PUBLICATIONS

Parts of this thesis have been published in prior work. The following gives a brief description of the origins of this work and what parts of this thesis it is a basis for.

The work in Chapter 3, related to the ESA-funded CSSP in particular, was developed as part of the CATSY project. It stems for the most part from the (non-public) project deliverables, which led to the publication of

V. Bos, H. Bruintjes, and S. Tonetta. “Catalogue of system and software properties”. In: *35th International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2016*. Proceedings. (Sept. 21–23, 2016). Ed. by A. Skavhaug, J. Guiochet, and F. Bitsch. Vol. 9922. LNCS. Trondheim, Norway: Springer, 2016, pp. 88–101. ISBN: 978-3-319-45476-4. DOI: 10.1007/978-3-319-45477-1\_8.

Chapter 4 is based on two prior publications which were made as part of the ESA-funded HASDEL project. During the initial phases, possible approaches to statistical model checking (SMC) were investigated, which led to the results published in D. Bohlender et al. “A review of statistical model checking pitfalls on real-time stochastic models”. In: *6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, ISOFA 2014*. Proceedings, Part II. (Oct. 8–11, 2014). Ed. by T. Margaria and B. Steffen. Vol. 8803. LNCS. Imperial, Corfu, Greece: Springer, 2014, pp. 177–192. ISBN: 978-3-662-45230-1. DOI: 10.1007/978-3-662-45231-8\_13.

Later during the project, a tool was implemented based on this work. After the project had completed, a case study was performed using it, which led to the publication of

H. Bruintjes, J.-P. Katoen, and D. Lesens. “A statistical approach for timed reachability in AADL models”. In: *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015*. (June 22–25, 2015). Rio de Janeiro, Brazil: IEEE Computer Society, 2015, pp. 81–88. ISBN: 978-1-4799-8629-3. DOI: 10.1109/DSN.2015.32.

The COMPASS toolset, described in Chapter 5, was revised as part of the COMPASS 3 project. The result was published previously as the following short tool paper by M. Bozzano et al. “The COMPASS 3.0 toolset (short paper)”. In: *Proceedings of the 5th International Symposium on Model-Based Safety and Assessment, IMBSA 2017*. 2017. URL: <https://drive.google.com/open?id=oB9DzO9PFFER2xRELXeGh6X1pjS1k>.

The work in Chapter 6 is based on the bachelor thesis:

L. Armbrorst. “Generating Simulink Models from AADL system descriptions”. Bachelor’s Thesis. RWTH Aachen University, Oct. 2015,

who under the supervision of the author both designed and implemented the Simulink translator and designed and constructed the SLIM and LEGO® models.

## 1.5 CONTRIBUTIONS

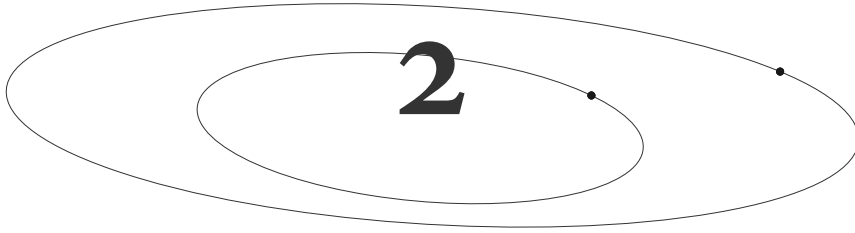
The work presented in this thesis contributed both theoretical in nature as well as practical applications that are present in the COMPASS toolset. They can be summarized as follows:

- » The SLIM language has been revised, bringing new features and capabilities. This includes extended support for timed specifications (time delay specifications, clocks in error models), the addition of event data ports, property specifications and state specifications. Aside from making the language easier to use and more powerful, these changes also bring it closer to the AADL. The fault injection system has been extended to support event inhibition and forced modes, giving more control over the influence of error models.
- » A taxonomy of requires types has been defined, permitting the identification of various design attributes from which such requirements can be derived, and possible methods of derivation. A design-attribute based approach is introduced as well, referred to as the CSSP.
- » The practical use of statistical model checking has been investigated, which lead to the identification of possible caveats and methods of classifying possible statistical model checking algorithms. A simulator has been implemented for the COMPASS toolset, allowing for the use of statistical model checking when verifying SLIM models, addressing the possible caveats by means of providing possible strategies to deal with them.
- » The COMPASS toolset has been updated and extended as part of the various ESA-funded projects. This includes the addition of support for non-deterministic models in the scope of performability analysis, the integration of statistical model checking (SMC), extended fault injection capabilities—

forced modes and inhibited events— and new property specification methods— properties can now be specified directly using SLIM (temporal) expressions, using patterns constructed from various parameters, or by means of the newly introduced CSSP.

- » A case study has been created, modeling (part of) the software running on the CubETH satellite, based on a prior case study that used the BIP framework. Model checking has been performed, verifying both properties derived from the structure of the model, based on architectural patterns, as well as those derived from requirement specifications. Note that no prior publications exist for this work.





## MODELING

*ABSTRACT – Similar to how any space project requires a system to be implemented to fulfill its requirements, any formal verification requires a model to check its properties. In this chapter, the modeling language used by COMPASS is introduced, namely SLIM. It has been derived from the AADL. First, an overview of the general syntax is given by which a model can be specified, followed by the semantics used by SLIM which provide the transition systems that the model represents.*



**V**ALIDATION and verification of any system model, in particular analyzing its requirements or reliability, strongly depend on the fact that the semantics of that model have been rigorously defined. To make this possible within the scope of COMPASS, the SLIM language has been created, which provides a modeling formalism suitable for — but not limited to — aerospace systems. SLIM has been derived from the Architecture Analysis & Design Language (AADL) (formerly avionics architecture description language), a language originating from, and standardized by, the automotive industry, and currently used in other domains as well.

The importance of a well-defined semantics for a language becomes apparent when deriving other artifacts from it, in particular those involved in reliability, availability, maintainability and safety (RAMS), failure detection, isolation and recovery (FDIR) or safety analysis. For such artifacts it is important that they are generated in a sound and complete manner, for which a clear semantics is required. For SLIM, such a semantics is defined, as will be explained in Section 2.5.

Since its inception, the SLIM language has seen multiple revisions, resulting from work performed in prior projects that followed COMPASS, for example AUTOGEF, HASDEL, and CATSY (cf. Section 5.1). The extensions and alterations from these projects have been brought together, ultimately leading to the current revision of

the language, SLIM 3.0. In the following, a basic overview of the language will be given.

## 2.1 SLIM AND MODELING AEROSPACE SYSTEMS

SLIM, short for System Level Integration Modeling, has been designed with modeling aerospace systems in mind. As such, it has some features specifically tailored to this domain, though they may be applicable to others as well. As aerospace systems need to be highly reliable, significant effort is put into making such systems fault tolerant, or resilient. This imposes two requirements on the modeling language: the ability to model fault-management systems, as well as the possible faults that may occur. The AADL architecture-based approach permits the prior, with the possibility to dynamically *reconfigure* the system as necessary. For the latter, SLIM features an error modeling extension, as will be explained in Section 2.3. In particular, the choice has been made to keep nominal and error models separate, and join them by means of a *fault injection*-based approach. This allows the nominal system to be developed independently from the error model.

A further aspect that needs to be considered is the refinement of systems as the development of a project progresses. This generally entails a hierarchical decomposition of systems into subsystems, at which point it has to be ensured that the subsystems fulfill the specifications of the system itself. The hierarchical nature of AADL permits to represent this. To ensure the consistency of the system specification after refinement, SLIM provides the notion of contracts, though this aspect will be discussed later in Section 3.3.3.

SLIM provides extensions on top of AADL to be able to specify the behavior of the system, in particular with respect to its data. Conditional behavior and effects on data can directly be specified on *states*, or *modes*, of the system.

In addition, SLIM features capabilities to model timed and hybrid systems. Aerospace systems show a clear need for such capabilities, be it to model, e.g., the charge state of a battery, or stringent time bounds on fault-management behavior. SLIM supports both fixed-rate clocks to model timing, as well as *continuous* variables which may change according to an arbitrary — but constant — rate.

## 2.2 MODELING SYSTEMS

In this section, the aspects of AADL that are relevant for SLIM are described. The full specification of AADL can be found in [AS5506C], that of SLIM in [SLIMv3].

AADL provides language features geared towards describing both hardware and software aspects of a system, such as buses, processors, threads and routines. Each AADL model is described in terms of components, the basic elements of the system it represents. Components are defined in terms of types and implementations. Types specify the interface of a component, by which it can communicate with

```

system gps_receiver
  features
    has_fix : out event port;
    lost_fix : out event port;
    signal : out data port int {Default => "100"};
    position : out event data port real;
end gps_receiver;

```

Listing 2.1: Component type for a GPS receiver.

other components. The implementation specifies the internal details, such as its subsystems, internal connections and possible modes of operation.

### 2.2.1 COMPONENT TYPES

An example of a component type is given in Listing 2.1. It represents a GPS receiver, with four ports that communicate its state. In this particular case, the component is defined as a **system** component. Alternatives are **bus**, **device**, **memory**, **process**, **processor**, **thread**, and **thread group**. Finally, **abstract** can be used for components for which the type is not defined.

#### Ports

Communication between components is achieved by means of ports, of which there are three kinds:

- » Event ports, which provide a means for communicating using instantaneous events. These events can be used to synchronize components as they can be used as transition triggers, which is explained further on;
- » Data ports, which provide continuous transmission of data. The type of data is specified by the port;
- » Event data ports, a combination of the above. Their behavior is similar to that of event ports, but an event is associated with some data.

Ports have a direction associated with them, being either *in*, *out* or *inout*. These directions mean respectively that a port can receive, transmit or both receive and transmit events respectively data. AADL further allows the specification of ordering and queuing of port transmissions. However, for the rest of this thesis this is not relevant, as SLIM employs its own semantics in this regard.

In the case of SLIM, port directions are limited to **in** and **out**, and— more importantly— communicate synchronously, meaning events and data are not queued, but take effect immediately.

In the example of Listing 2.1, the `has_fix` and `lost_fix` ports both provide the ability to synchronize (trigger) on these events (acquiring respectively losing a GPS fix). The `signal` port provides a continuously readable source of data, in this case

```

system implementation gps_receiver.impl
  states
    init : activation state urgent in 4.0 sec;
    acquire: state urgent in 12.0 sec;
    enabled: state urgent in 1.0 sec;
    failed: state;
  transitions
    init -[when signal > 0 within 2.0 sec to 4.0 sec]-> acquire;
    init -[when signal = 0]-> failed;

    acquire -[has_fix when signal > 0
              within 1.0 sec to 12.0 sec]-> enabled;
    enabled -[position(read_pos(signal)) when signal >= 25
              within 1.0 sec to 1.0 sec]-> enabled;
    enabled -[lost_fix when signal < 25 and signal > 0]-> acquire;
    enabled -[when signal = 0]-> failed;
end gps_receiver.impl;

```

Listing 2.2: Component implementation for a GPS receiver.

the signal strength represented as an integer. Finally, the event data port `position` provides an event which, upon synchronization, allows the position to be read as a real number.

### 2.2.2 COMPONENT IMPLEMENTATIONS

The actual details of how a component should behave are defined by a component implementation, an example of which is given in Listing 2.2. Component implementations can describe their own configuration by means of *modes*, or state by means of *states*. Changes to modes and states are described by *transitions*. Furthermore, component implementations can also specify that other components are contained inside by means of *subcomponents*, and *connections* between those, itself and its environment.

In the example, a state machine is constructed by defining four states and six transitions, which describe how the GPS receiver acquires a fix for its position.

#### *Modes and States*

Modes and states describe the possible configurations or states of the system. The difference will be described more in depth when treating transitions, but, in short, modes control the topology of the system, whereas states control its behavior from the bottom up.

The states given in the example show two important attributes: which state is active upon initialization, and possible invariants. Each component implementation must

specify at least one *initial* or *activation* state. In the initial state of the entire system, these are the modes and states that are active. However, their difference becomes apparent when a component is *reactivated*: upon reactivation, components with an **activation** mode or state will have this state become active. When they have an **initial** mode or state, the previous mode and state remains active.

Invariants are specified by means of a **while** or **during** keyword. The invariant specifies an expression over clocks or continuous variables which is guaranteed to hold true while its respective mode or state is active. In case of the example, the **during** statements specify an upper bound on the amount of time the system is allowed to reside in that state. It implies that when this upper bound is reached, the system must transition to another mode before time can move forward again.

Invariants may also specify trajectory equations, which are expressions of the form  $x' := a$ , where  $x$  is a continuous variable, and  $a$  some constant real number. Such trajectory equations specify the rate at which  $x$  changes while the mode for which it is defined is active (i.e.,  $\dot{x} = a$ , with  $a \in \mathbb{R}$ ). For example, such trajectory equations can be used to specify the discharge rate of a battery under various conditions.

### Transitions

Modes can be switched by means of transitions. Such transitions specify a source  $m$  and target  $m'$ , and are written in the form  $m \text{ --} [ t \text{ when } g \text{ between } l \text{ and } u \text{ then } e ] \text{ --} m'$ . Here,  $t$  represents the trigger. The trigger is an event or event data port that synchronizes the transition with any other such ports that may be connected. For modes, such triggers may only *receive* events, whereas for states they may also *send* them. Furthermore, states also permit the trigger not to be specified. In that case, the transition becomes an *internal* transition, which is not visible to the environment of the component it is defined in.

A transition may optionally specify a *guard*, represented as **when**  $g$ , which is a Boolean expression  $g$  over the data variables readable within the component. If at the time of receiving the event the expression evaluates to true, the transition is said to be enabled. Otherwise, the transition cannot be taken, possibly blocking any other synchronizing transition. If no guard is specified, it is implicitly assumed to always be enabled. A shorthand to bound the time during which a transition is enabled is provided by means of the **between**  $l$  **and**  $u$  syntax. Here,  $l$  and  $u$  specify a (numeric) lower and upper bound respectively, during which the transition is enabled.

Finally, **then**  $e$  allows (optionally) effects to be specified, where  $e$  is a set of expressions of the form  $x := f$ , which assign the result of expression  $f$  to data port or component  $x$ . The value of  $x$  will be updated after the transition has been taken, such that effects like  $x := y$ ;  $y := x$  simply swap values.

A special transition exists to allow fine grained control over the *reactivation* of a component (see Section 2.2.3). Normally, reactivation behavior is defined by specifying either an **activation** or **initial** mode. With reactivation transitions, this

can be specified depending on the current state of the component. A reactivation transition is a transition of the form  $m \text{ --[ @activation then } e \text{ ]--> } m'$ . Such a transition is triggered if the component is deactivated while in mode  $m$ , and subsequently activated again. The component will then transition to mode  $m'$  while being activated. Thus, if such a transition is specified and enabled, it will override any **activation** mode. Any effects specified by  $e$  will apply as well.

**Time delays** The time delays that can be specified by means of the **urgent in** and **between  $l$  and  $u$**  keywords are internally translated into invariants and guards. If a component contains such specifications, a new clock is introduced. This clock is used to specify an upper bound in invariants for modes which have a **urgent in** specification, and specify a guard with a lower and upper bound for transitions with a **between  $l$  and  $u$**  specification. Finally, on each transition the clock is reset to 0. For details see Section 2.5.1.

### 2.2.3 COMPOSITION AND RECONFIGURATION

Models of systems are constructed by composing various subsystems into larger systems. In SLIM, this is done by means of specifying subcomponents, and connecting them. In the following, the example shown in Listing 2.3 is used, which is a system consisting of two GPS receiver components are presented before. Both components run in a redundant configuration, with the containing component forwarding the position of the first GPS receiver that got a fix.

#### *Subcomponents*

SLIM supports two kinds of subcomponent: *data* subcomponents, and *non-data* subcomponents. Data subcomponents are simple data elements based on either an intrinsic SLIM data type, or a user-defined data component. They permit values to be read from and written to by means of transition guards and effects. In the GPS example, two data subcomponents are defined (e.g., `has_fix_prim`) which are used to store a Boolean value. Guards such as **when not** `has_fix_prim` and effects like **then** `has_fix_prim := false` are used to read from respectively write to them.

Non-data subcomponents define the topology of the entire model. They reference other components (though never recursively) instances of which will be contained by the component. Components that do not have any non-data subcomponents are said to be *atomic* — also named *leaf* components — whereas those that do are labeled *composite*. In the example of Listing 2.3, `primary` and `secondary` define two subcomponents based on the GPS receiver component.

#### *Connections*

In order to allow subcomponents to interact with each other and their environment (parent), connections can be made between the ports of components. Port connection can be made between ports of the same kind, and for (event) data ports

```

system gps
  features
    fix : out data port bool;
    position : out event data port real;
end gps;

system implementation gps.impl
  subcomponents
    primary : system gps_receiver.impl;
    secondary : system gps_receiver.impl;

    has_fix_prim : data bool {Default => "false"};
    has_fix_sec : data bool {Default => "false"};
  connections
    flow true -> fix in modes(m_prim, m_sec);
    flow false -> fix in modes(m_none);
    port primary.position -> position in modes(m_prim);
    port secondary.position -> position in modes(m_sec);
  modes
    m_none : activation mode;
    m_prim : mode;
    m_sec : mode;
  transitions
    m_none -[primary.has_fix then has_fix_prim := true]-> m_prim;
    m_none -[secondary.has_fix then has_fix_sec := true]-> m_sec;

    m_prim -[primary.lost_fix when not has_fix_sec
              then has_fix_prim := false]-> m_none;
    m_prim -[primary.lost_fix when has_fix_sec
              then has_fix_prim := false]-> m_sec;
    m_prim -[secondary.has_fix then has_fix_sec := true]-> m_prim;
    m_prim -[secondary.lost_fix then has_fix_sec := false]-> m_prim;

    m_sec -[secondary.lost_fix when not has_fix_prim
              then has_fix_sec := false]-> m_none;
    m_sec -[secondary.lost_fix when has_fix_prim
              then has_fix_sec := false]-> m_prim;
    m_sec -[primary.has_fix then has_fix_sec := true]-> m_sec;
    m_sec -[primary.lost_fix then has_fix_sec := false]-> m_sec;

end gps.impl;

```

Listing 2.3: Component type and implementation of a composite GPS system.

between those of the same type. Such connections permit events and data to flow through the system from one component to the other. These connections are always directional. Between systems, output ports must be connected to input ports. From subsystem to system, connections can be made between two output ports, and in the other direction — toward the subsystem — between input ports. In SLIM, some restrictions have to be adhered to in order to avoid cyclic behavior. In particular, event ports may not be forwarded to an output port from an input port of the same system. Connections are written in the form **port** *p* -> *q*; , where *p* and *q* refer to the respective ports to connect.

When a connection is made between event ports, all components involved in the transitive closure of the active connection have to participate in synchronizing on the event. In other words, if one component cannot make a transition based on this event, the event is *blocked*.

A connection between data ports allows the instantaneous transmission of data between components. As soon as data is assigned to the input of such a connection, all connected data ports permit this value to be read. More interesting is the data *flow*, which allows the input of a data connection to be an expression over the readable data in the component. This allows the data to be transformed as it is propagated through the connections. For this, the syntax **flow** *e* -> *p*; is used, where *e* is an expression of the same type as port *p*.

### *Bindings*

Subcomponents can be *bound* to each other. By specifying a binding, a relation between subcomponents is made explicit<sup>1</sup>. Three such bindings are possible:

1. *running on*: This binding specifies for a given **process** component the **processor** component it runs on.
2. *stored in*: This binding specifies for a given **process** the **memory** component it is stored in.
3. *accesses*: This binding specifies for a **processor**, **memory**, **bus**, **device** or **system** component the (common) **bus** component it accesses.

These bindings influence how errors may propagate within the system, which will be discussed later in Section 2.4.2.

### *Reconfiguration*

As previously mentioned, modes play an important role in the configuration of the system. In particular, using modes the topology can be reconfigured dynamically. This is done by specifying for components and connection in which modes they are active, using the **in modes** (...) syntax.

<sup>1</sup> Prior versions of SLIM required bindings to be specified to make port connections between subcomponents, however, this restriction has been lifted.

```

package Gps
  public
    system gps
    ...
  end gps;

  system implementation gps.impl
  ...
  end gps.impl;
private
  system gps_receiver
  ...
  end gps_receiver;

  system implementation gps_receiver.impl
  ...
  end gps_receiver.impl;
end Gps;

```

Listing 2.4: Example package containing the GPS and GPS receiver components.

If for an element it is not specified for which modes it is active, it is active in all modes. Otherwise, it is active only in those modes which have been listed. In the GPS example, take for instance the connections targeting the `position` port. The `position` event from the `primary` respectively `secondary` component is forwarded when the `m_prim` respectively `m_sec` mode is active. Note that neither connection is enabled in the `m_none` mode.

For data subcomponents, reactivation — that is, activation after having been deactivated — will reset the value of the component to its default, if specified. When reactivating non-data subcomponents: If an **initial** mode is specified, the component restores its prior state. Otherwise, it returns to the **activation** mode, and any data subcomponents will be reactivated. Non-data subcomponents will be reactivated in either case. Data components will be reset to their initial value (if any) when an **activation** mode is specified, unless overridden by the effects of a reactivation transition.

#### 2.2.4 PACKAGES

In order to help structure larger models, components can be defined in packages, which provides namespacing support. Packages consist of a public section and an optional private section. Between packages, only components from a public section are accessible. Within a private package, private components are accessible as well. An example is given in Listing 2.4, where the `gps` component is declared public, and the `gps_receiver` component is declared private.

```

data Position
end Position;

```

```

data Position.Vec3
subcomponents
  x: data real;
  y: data real;
  z: data real;
end Position.Vec3;

```

Listing 2.5: Example of a user-defined data component, both an abstract type and a concrete implementation.

### 2.2.5 DATA

SLIM provides a number of built-in types, as well as the ability for users to define custom types by defining data components. These types can be used when defining (event) data ports or data subcomponents, which can subsequently be used in flow, invariant, guard and effect expressions.

The SLIM provided built-in types are as follows:

- » **bool**: Provides the values **true** ( $\top$ ) and **false** ( $\perp$ );
- » **int**: Provides values in  $\mathbb{Z}$ ;
- »  $[l \dots u]$ : Specifies a range of values  $\{x \mid x \in \mathbb{Z} \wedge x \geq l \wedge x \leq u\}$ , with  $l, u \in \mathbb{Z}$  and  $l \leq u$ ;
- » **real**: Provides values in  $\mathbb{R}$ ;
- » **clock**: Provides values in  $\mathbb{R}_{\geq 0}$  that continuously increase over time with a fixed rate;
- » **continuous**: Provides values in  $\mathbb{R}$  that continuously increase over time with a user specified rate (using trajectory equations).

More complex types can be specified using tuples, denoted as  $(t_0, t_1, \dots, t_n)$ , where  $t_i$  is a valid built-in or tuple type in SLIM.

Alternatively, keeping in line with AADL, components of the **data** category can be specified as well. The example in Listing 2.5 presents a type `Position`, which in terms of SLIM represents an *abstract* type. Variables of such abstract types hold some arbitrary value which is not concretely specified and can only be tested for (in)equality or assigned to other variables. If an implementation is given, it can be mapped to a composite concrete type. The `Position.Vec3` implementation in the example represents a SLIM tuple consisting of three real numbers, i.e., **(real, real, real)**.

SLIM provides the package `SLIMdatatypes`, which contains aliases for the built-in types as data component types. These can be used in lieu of SLIM types, e.g.,

```

package Gps
  public
    ...
  properties
    Constants => "
      signal_max : int := 100;
      greenwich : Position;
      read_pos : function int -> real;
    ";
end Gps;

```

Listing 2.6: Example constants specification for a package.

SLIM datatypes: **bool** instead of simply **bool**, and are more compatible with AADL.

### 2.2.6 CONSTANTS

In order to avoid hardcoding recurring data values, SLIM provides an extension that allows user-defined constants to be specified. Such constants can be used in expressions, but also range types, trajectory equations and error rates (the latter are explained in Section 2.3). These constants are specified by means of a special property `Constants` for packages. An example is given in Listing 2.6.

Two types of constants are defined in this example. The first, `signal_max`, is a constant integer value of 100. The second is an example of an abstract constant. Defined for the abstract type `Position` (see Section 2.2.5), it defines some arbitrary but fixed value for this data type.

The third element, `read_pos`, defines a so-called *uninterpreted function*. This is an abstract function that accepts some arguments as input, and results in some value as output. The relation, however, is left undefined (for more details see Section 2.5.3).

### 2.2.7 HYBRID MODELS

The `clock` and `continuous` types make it possible to specify *timed* models, which allow reasoning over the duration of certain behavior. This means SLIM effectively provides two kinds of behavior: *discrete* and *timed*. Models that exhibit both kinds of behavior are called *hybrid*.

Discrete behavior can be described in discrete steps, whereas timed behavior is described as a continuous evolution. Discrete behavior is based on the transitions between modes and states in the model. At the moment such transitions are taken, both the discrete and continuous values of the model are updated according to the transitions' effects, based on constant values. Continuous behavior is described solely by the progression of time, at which point the discrete aspects of the model

```

property set SLIMpropset is
  SlimExpr: type aadlstring;
  Default: SlimExpr applies to (port, data);
  ClockTimeUnit: type enumeration (Milliseconds, Seconds,
                                     Minutes, Hours, Days);
  TimeUnit: ClockTimeUnit applies to (data);
  FDIR: aadlboolean => false applies to (all);
  Alarm: aadlboolean => false applies to (port);
  Observable: aadlboolean => false applies to (port);
  Max_Aadlinteger : constant aadlinteger => 2#1#e32;
  ...
end SLIMpropset;

```

Listing 2.7: Example property set, based on a subset of the built-in SLIMpropset.

remain unchanged. The continuous values are updated according to the (unique) solution of the linear equation system that describes the continuous variables.

SLIM permits timed models to be written either without the use of *time units*, such as **sec**, **min** or **hour**, or with *time units* fully specified. The example for Listing 2.2 for instance makes use of such time units. Internally, these time units will be translated, by means of scaling, into a single abstract notion of time. Converting between timed values from **clock** data to, e.g., a **real** data component is possible by specifying the time unit needed to represent the value. For example, an effect such as  $x := c \text{ sec}$  will store the time since clock  $c$  was last reset into variable  $x$ , expressed in seconds. The converse works similarly.

### 2.2.8 PROPERTIES

An aspect introduced in the latest version of SLIM — 3.0 — is the ability to specify *properties*. Properties permit the user to specify the value for certain attributes for all elements in the model, ranging from components down to individual modes.

Properties are defined in *property sets*. Such property sets can be thought of as packages for properties. Property sets can define new property types, constants for property values and the properties which may be used in the model.

Property types are used to limit the range of values a given property may be assigned, similar to the built-in data types supported by SLIM<sup>2</sup>. Various basic property types exist, which can be aggregated into more complex types, or simply be aliased. SLIM supports the following types:

- » **aadlboolean**: simple Boolean type;
- » **aadlinteger**: simple integer type, optionally bound by a range;
- » **aadlreal**: simple floating point type;

<sup>2</sup>This similarity is addressed in Section 2.7.1.

- » **aadlstring**: simple string type;
- » **enumeration**: enumeration of identifiers;
- » **reference**: A reference to an element in the model. Can optionally be restricted to various subsets of element types (e.g., ports or modes);
- » **classifier**: A reference to a classifier in the mode. Can optionally be restricted to various classifier types (e.g., **device** or **system**);
- » **range**: allows values that represent a range of integers or real numbers;
- » **list of**: a list of property values of a given type;
- » **record**: a record containing named key-value pairs.

In addition, numeric types may have an additional unit specified, which consists of a base unit, and derived units. For example, time is generally represented in seconds, with derived units for minutes and hours.

By defining a new type, it is possible to specify further properties and property value constants using that type to limit their domain. In the example given in Listing 2.7, the property type `SlimExpr` is an alias for **aadlstring**, which SLIM uses internally to represent expressions (such as for default values).

Properties are defined in terms of an identifier and type for its value domain. Additionally, it may specify for what model elements it is applicable. In the example above, the property `Default` has the type `SlimExpr` and applies only to data ports and components.

Finally, property constants can be used to define often recurring values, such as for example some limit. In the above example, the constant `Max_Aadlinteger` of type **aadlinteger** is defined with a value of  $2^{32}$ .

Using properties in a model is done by means of property *associations*. Such associations assign a property value to some property that can be specified for a particular element in the model. Recall from the example in Listing 2.2 the data subcomponent definition

```
has_fix_prim : data bool {Default => "false"};
```

Here, the `Default` property is assigned the value `"false"`. Alternatively, elements that are contained by some other element may be assigned a value by the parent element using the **applies to** syntax. For example, one could write

```
system implementation gps_receiver.impl
...
subcomponents
  has_fix_prim : data bool
...
properties
  Default => "false" applies to reference(has_fix_prim);
```

## 2.3 ERROR MODELING

In any physical system, faults may occur due to a variety of causes (wear, single event upsets (SEUs), damage). Such faults have to be modeled as well in order to perform any analysis on the model related to, e.g., fault management and reliability. One possibility is to model such faults into the model, but generally the two domains are treated separately (and in fact developed and analyzed by different engineers). SLIM allows error models to be defined separately, which can be integrated with the nominal model at a later state (see Section 2.4).

SLIM error models describe how fault may evolve and propagate throughout the system. Similar to modes and transitions in nominal models, error states and transitions between them describe the exact error behavior. Error events and propagations then trigger such transitions. An example error model is given in Listing 2.8.

Error models are described in terms of types and implementations akin to nominal components. As with nominal components, communication is possible between error components, though in this setting the intent is to model the propagation of errors in the system. Such *error propagations* model the fact that an error in a single component can trigger an error in a neighboring component.

The possible propagations are defined in error model types (similar to nominal event ports). Propagations are given a direction, with the same meaning as ports in nominal component. For details on how propagations are connected/mapped, see Section 2.4.

Error *events* are events that occur internally in the error component, and are defined in error model implementations. Error events may be associated with an *error rate*, which describes the probabilistic distribution by which the events occur. It is based on the exponential distribution function  $\lambda e^{-\lambda t}$ , where  $\lambda \in \mathbb{R}_{>0}$  is the rate by which the error event occurs. For a given time period  $t$ , the total probability of the event occurring is  $1 - e^{-\lambda t}$ . Its expected value is  $E[X] = \frac{1}{\lambda}$ , where  $X$  is the error event associated with rate  $\lambda$ . In the example of Listing 2.8, the `trans_fail` event is given a rate of 0.5 per hour, meaning it is expected to occur once every two hours, and the probability of it occurring after 30 minutes is  $1 - e^{-0.5 \times 0.5} \approx 0.221$ .

Error model implementations describe the behavior of the error model in terms of error states and transitions between such states. The transitions between the states are triggered by an error propagation, making communication between error components possible; an error event or the **reset** event. The **reset** event can be triggered by a nominal component, and allows the nominal component to attempt a recovery of an error condition by changing the error state. Note that the **reset** event is the only means by which the nominal components can influence an associated error model directly.

Error model implementations may also specify clocks, which can be used to model the delayed nature of certain errors. Similar to nominal components, error states

```

error model err_timed
  features
    ext_err : in error propagation;
end err_timed;

error model implementation err_timed.impl
  events
    trans_fail : error event occurrence poisson 0.5 per hour;
    hot_fail : error event occurrence poisson 0.05 per day;
    perm_fail : error event occurrence poisson 0.001 per day;
    trans_clear : error event;
    perm_reset : error event;
  states
    -- allowed states
    e_nom : initial state;
    e_trans : error state urgent in 2.0 sec;
    e_hot : error state;
    e_perm : error state;
  transitions
    e_nom -[trans_fail]-> e_trans;
    e_trans -[trans_clear within 0.5 sec to 2.0 sec]-> e_nom;

    e_nom -[hot_fail]-> e_hot;
    e_nom -[ext_err]-> e_hot;
    e_hot -[@activation]-> e_nom;

    e_nom -[perm_fail]-> e_perm;
end err_timed.impl;

```

Listing 2.8: Example error model and error model implementation showing transient, hot and permanent errors.

may specify invariants for these clocks, and error transitions may specify guards. Again, time delays can be used as well.

The example shown in Listing 2.8 shows three kinds of error behavior according to their lifetime: The `trans_fail` event defines a *transient* error, that will return to the original state after a delay of maximally 2 seconds. The `perm_fail` event defines a *permanent* error, which cannot be recovered from (cf. the **initial** error state). Finally, the `hot_fail` defines an *hot* error, but can be recovered from by interacting with the system, in this case reactivating it (as this will trigger the reactivation transition).

One important remark is that the use of probabilistic events cannot be combined with timed transitions, i.e., such an event cannot be used as the trigger of a time-delayed transition. The reason is that the semantics would not be well defined:

```

system implementation gps_receiver.impl
  states
    ...
  properties
    ErrorModel => classifier(err_timed.impl);
    FaultEffects => (
      [State=>"e_trans"; Target=>reference(signal); Effect=>"10"];],
      [State=>"e_hot"; Target=>reference(signal); Effect=>"0"];],
      [State=>"e_perm"; Target=>reference(signal); Effect=>"0"];]
    );
end gps_receiver.impl;

```

Listing 2.9: Example fault injection properties.

The exponential probability distribution has an infinite support, such that its delay always falls in the interval  $\mathbb{R}_{\geq 0}$ . This directly conflicts with any bounded interval based on a time delay.

A further restriction is that for every transition  $e \xrightarrow{[t \dots]} e'$ , the pair  $(e, t)$  has to be unique within the error model implementation. The reason is technical in nature, as it facilitates the support for fault effects as described in Section 2.4.3.

## 2.4 MODEL EXTENSION

After both nominal and error components have been defined, they can be linked together using the process of *model extension*. This process enables the specification of the effects the error model has on the nominal, as well as provide the link for the **reset** event.

First, it is defined which error models are linked, or associated, to which nominal components. This is specified by means of the `ErrorModel` property, which names the error model classifier that is to be associated with the component for which the property is specified. The latter can be a component type, component implementation or subcomponent (the more specific property association overriding any other). For an example, see Listing 2.9, which associates `err_timed.impl` with `gps_receiver.impl`.

Specifying an error model will make the necessary associations, which will allow the error model to run in parallel with the nominal one, providing error propagations to the full system and making it possible for the nominal model to send **reset** events. However, the error model will not have any direct influence on the nominal model this way. For this, *fault injections* will have to be specified.

### 2.4.1 FAULT INJECTIONS

Three types of fault injections can be specified:

- » fault effects;
- » forced modes;
- » event inhibition.

*Fault effects* specify how the various error states affect the data of the component to which they are associated. Each effect specifies the state it applies to, and the expression which determines how the data is affected. An example is shown in Listing 2.9. Three fault effects are specified, which affect the signal data port. When the `e_trans` error state is active, the port's value is forced to 10 (which in turn will trigger the nominal model to try and reacquire it). The two other effects apply to the states `e_hot` and `e_perm`, force the signal value to 0, which subsequently triggers the nominal component to enter the failed mode.

*Forced modes* provide for any given error state the possible modes the nominal component is allowed to be in. If the error state becomes active, and transitions leaving this set of modes are disabled. If the current mode of the nominal component is not in this list, it is forced to the first listed mode. For example, a forced mode specification of

```
ForcedModes => ([State=>"e_perm"; Modes=>(reference(failed))]);
```

would force the nominal component to enter, and remain in, the failed mode as long as the error model is in the `e_perm` state.

*Event inhibition* determines which effects can no longer be triggered or triggered upon when the state for which it is specified is active. This makes it possible to alter the behavior of the component upon entering an error condition. As an example,

```
Inhibit => ([State=>"e_hot"; Ports=>(reference(position))]);
```

would block the sending of the `position` event while the `e_hot` state is active.

#### 2.4.2 ERROR PROPAGATIONS

The model extension process takes into account how error propagations may take place in the system topology. Unlike nominal ports, which are connected by explicitly specifying port connections, propagation ports are linked by name when performing model extension.

Propagations are linked automatically between subcomponents and their parents, and in some cases between subcomponents. For this to apply, both the names of the propagations have to match, as well as both propagations having a consistent direction (e.g., an **in** propagation is not connected to an **out** propagation). Similar topological restrictions apply as for nominal ports.

Not all (sub)components with an associated error model will have their propagations connected however. Between subcomponents, some binding has to exist in order for the connection to be made (otherwise, unrelated components would be

able to influence each other). Thus, if either an *accesses, running on, or stored in* binding is specified, the subcomponents involved in the binding are connected with each other.

### 2.4.3 MODEL TRANSFORMATION

The model extension process itself is performed by transforming the input SLIM model. In short, this transformation adds the error components as subcomponents into the system, and modifies the nominal components to behave accordingly.

#### *Generating Error Systems*

Each used error model is translated into a **system** component, which is added as a subcomponent to the implementation it is associated with. The interface of this system, i.e., its ports, is based on the defined propagations, but also the reset event, the error events, and a data port that transmits the current error state. For each of these items, a port is added. The error event ports and data port are required, despite not influencing other error models, in order to apply the fault injections, if any, to the nominal component.

As both the type and implementation of the error model define its interface, a unique system component implementation *and* type are generated for each error model implementation (as opposed to, say, mapping an error model type to system type). The system type is generated is follows:

- » Each error propagation with direction  $d$  is mapped to a new event port with the same direction  $d$ .
- » Each error event is mapped to an outgoing event port, with a possible annotation for the error rate.
- » An input event port for the **reset** event is added.
- » An outgoing data port `errorState` is generated, of which the type is an enumeration, with a unique identifier for each error state defined in the implementation.

The translation of the error model implementation to a system implementation is fairly straightforward:

- » Each clock is mapped to a clock data subcomponent.
- » Each error state is mapped to a mode, with **initial** or **activation** states mapped to the corresponding modes, and respecting any invariants specified.
- » Each transition is translated directly, with the source and target state translated to their mode counterparts. However, to each transition  $m \rightarrow m'$  an effect `errorState := m'` is added, which updates the data port that tracks the current error state. The same holds for reactivation transitions.

- » The **reset** event is made input-enabled. For modes  $m$  that do not already have an outgoing transition triggered by **reset**, a transition  $m \text{ --}[\text{reset}] \text{ -->} m$  is added.

### *Integrating Error Systems*

For each component with an associated error model, a new subcomponent will be added that is based on this error component.

The nominal component is modified to synchronize on the error events and propagations. For each event and propagation, a self-loop is added to each mode that is triggered on this event or propagation. Furthermore, ports are added to the component's type for each error propagation, and they are connected to the error subcomponent.

Connections are made between nominal components to accommodate for the possible error propagations. Such connections are made between components and their subcomponents that contain an error subcomponent, thus making the error propagation available on its interface. Furthermore, error propagations are connected between subcomponents that share an *accesses*, *running on*, or *stored in* relation.

As error propagations will be integrated into the interface of the nominal component, a unique type and implementation is generated for every combination of nominal component and error type. Starting from the original type, the following modifications are made:

- » For each error propagation, a corresponding event port is added.
- » A data port is added matching the type of the data port in the error component representing the current error state.

The implementation requires more extensive modifications, in particular related to error propagations. For the implementation the modifications look as follows:

- » The generated error implementation is added as a subcomponent.
- » A self-loop is added to each mode that is triggered on one of the incoming error propagation ports, outgoing error propagation ports of the error subcomponent, or error event ports of the error subcomponent (ensuring the error state stays synchronized).
- » A connection is added from the error subcomponent's `errorState` port to the `errorState` port of the component itself<sup>3</sup>.
- » For each error propagation port of the error subcomponent, a connection is added to the corresponding port of the component itself.

---

<sup>3</sup>As SLIM prohibits transitions from reading data of subcomponents, this connection is required.

Further propagation connections for subcomponents are made, based on whether or not such a subcomponent has an associated error model, and if there exists a binding between these subcomponents. Subcomponents that do not have an error model associated are not considered. Otherwise, the following rules are applied to generate propagation connections:

- » Propagations from the environment to subcomponents: If a subcomponent contains an error propagation port with the same name as a propagation port of the component it is contained in, and they have the same direction, they are connected.
- » Propagations between the component and its subcomponents: If a subcomponent contains an error propagation port with the same name as a propagation port of the error subcomponent of the component it is contained in, and they have opposing directions, they are connected from out- to in-port.
- » Between any two subcomponents that share a binding: If both subcomponents contain an error propagation port with the same name, and they have opposing directions, they are connected from out- to in-port.

#### *Adding Fault Injections*

After the error components have been integrated into the model, any specified fault injections are applied. These fault injections are applied in the following order:

1. forced modes;
2. fault effects;
3. event inhibition.

**Forced modes** For forced modes, two modifications are made: The transitions leaving a nominal mode are guarded such that they are disabled when they leave the set of current forced modes, if any. For modes that are outside this set, a self-loop that is triggered by the current error event or propagation is replaced by a transition to the first mode of the set of forced modes that will be active in the new error state.

The following procedure is used to apply the forced modes to the nominal component:

- » For every mode  $m$  in the component, the disjunction over every comparison  $\text{errorState} = e$  for each error state  $e$  for which  $m$  is a permitted mode, is conjoined to the invariant of  $m$ . This prevents  $m$  becoming the active mode while this is prohibited.
- » For every mode  $m$  in the component, a transition is added to the first mode  $m_f$  in the forced mode-lists specified for error states  $e$ , with the guard  $\text{errorState} = e$ , providing a transition to the first forced mode of an activated error state.

The invariants added will ensure the system enters and remains in any forced mode when applicable. The transitions added ensure the system can and will enter a forced mode when an error state with forced modes specified becomes active.

**Fault effects** For fault effects that apply to a data component that is not the target of a data flow, an effect is added to each transition in the nominal component that is triggered by the error event that leads up to the error state for which the effect applies. For data components that are a target of a data flow, the flow is modified such that under no error condition the original source expression is used, and when an error effect applies the effect's expression is used. To this end, a **case** expression is used.

An important restriction should be considered for fault effects. These effects are both applied when an error event or propagation occurs, or when a nominal transition occurs. The latter can simply check the current error state and apply it accordingly. The prior, however, will have to determine the next error state (that the effect or propagation leads to) based on the current state and the error event or propagation. For this reason, an error model may not define two transitions leaving the same error state with the same error event or propagation.

Fault effects are added to each mode  $m$  in the component, by modifying existing transitions or flows. Assuming a fault effect targeting a data port or component  $d$ , applying value  $v_f$  and active in error state  $e$ , the following modifications are made:

- » Each flow **flow**  $a \rightarrow d$ ; is replaced by a flow  
**flow case**  $\text{errorState} = e \text{ then } v_f \text{ otherwise } a \text{ end}; \rightarrow d$ ;
- » For each transition  $m \text{ } \text{--}[t \text{ when } g \text{ then } f]\text{--} \rightarrow m'$ , including reactivation transitions, where  $t$  is *not* an error propagation, error event or **reset**, the assignment  $d := \text{case errorState} = e \text{ then } v_f \text{ otherwise } a \text{ end};$  is added to  $f$ , where  $a := ex$  if  $d := ex \in f$ , and  $a := d$  otherwise.
- » For each transition  $m \text{ } \text{--}[t \text{ when } g \text{ then } f]\text{--} \rightarrow m'$ , where  $t$  is an error propagation, error event or **reset**, the assignment  $d := \text{case errorState} = e' \text{ then } v_f \text{ otherwise } a \text{ end};$  is added to  $f$ , where  $a := ex$  if  $d := ex \in f$ , and  $a := d$  otherwise. Here,  $e'$  is the error state *preceding*  $e$ . This error state can be determined uniquely, as the combination of error state and error event or propagation is unique. This is necessary, as the value of `errorEvent` will only be updated *after* the transition has been taken.

**Event inhibition** Finally, event inhibitions are applied by adding an extra guard to transitions triggered by an inhibited event, that disables them when such an inhibition comes into effect. The procedure is straightforward. For each effect inhibition specified for error state  $e$  and event port  $p$ , and each transition  $m \text{ } \text{--}[p \text{ when } g \dots]\text{--} \rightarrow m'$ , the guard  $g$  is replaced by **and**  $\text{errorState} := e$ .

## 2.5 SEMANTICS

As previously mentioned, an important aspect of SLIM is its semantics. Whereas the SLIM models are described in terms of components, the semantics of those models are expressed using event data automata (EDAs), and their interactions, resulting in a network of event data automata (NEDA).

### 2.5.1 EDAs

An EDA is a tuple of the form

$$\mathcal{A} = (M, m_0, X, v_0, \chi, \phi, E, \rightarrow, \Rightarrow),$$

where

- »  $M$  is a finite set of modes;
- »  $m_0 \in M$  is the initial mode;
- »  $X$  is a finite set of variables, subdivided into the disjoint sets
  - $IX$ , the input variables;
  - $OX$ , the output variables;
  - $LX$ , the local variables;
- »  $v_0 \in V_X$  is the initial valuation, where  $V_X$  is the set of all *valuations*, which are partial functions that assign values to the variables in  $X$ ;
- »  $\chi : M \rightarrow (V_{LX} \rightarrow \mathbb{B})$  specifies the *mode constraints* and associated with every mode a function that maps the valuation of local variables to Boolean values.
- »  $\phi : M \rightarrow (LX \rightarrow \mathbb{R})$  specifies the change rates, and associated with every mode a function that maps local variables to real numbers;
- »  $E$  is a finite set of events, subdivided into the disjoint sets
  - $IE$ , the input events;
  - $OE$ , the output events;
- »  $\rightarrow \subseteq M \times E_\tau \times (\text{Exp}(LX \cup IX) \cup \{\varepsilon\}) \times (V_X \rightarrow \mathbb{B}) \times (V_X \rightarrow V_X) \times M$  is a finite *transition relation*, of which elements are denoted as  $m \xrightarrow{e, ex, g, f} m'$ . Here,  $E_\tau = E \cup \{\tau\}$ .
- »  $\Rightarrow \subseteq M \times (V_X \rightarrow V_X) \times M$  is a finite *reactivation transition relation*, of which elements are denoted as  $m \xrightarrow{f} m'$ .

Valuations from  $V_X$  map variables to their respective values. Thus,  $\chi$  specifies functions that evaluate to  $\top$  or  $\perp$  depending on the current valuation. Furthermore, for a mode transition  $m \xrightarrow{e, ex, g, f} m'$ ,  $m$  and  $m'$  are the *source* and *target* mode respectively.  $g$  specifies a *guard* which evaluates to  $\top$  or  $\perp$  depending on the current

valuation, and  $f$  the *effect*, which maps one valuation to another. Furthermore,  $ex$  is an *expression*  $\text{Exp}(X)$  over variables in  $X$ , or the empty expression  $\varepsilon$ , representing the trigger value for event data ports. For a reactivation transition  $m \xrightarrow{f} m'$ ,  $m$ ,  $m'$  and  $f$  hold the same meaning.

The function  $\text{Exp}(X)$  denotes the expressions over variables in  $X$ , which are functions  $(V_X \rightarrow (\mathbb{B} \cup \mathbb{E} \cup \mathbb{R}))$  that are the result of applying an expression over variables in  $X$  given a valuation from  $V_X$ , which result in a Boolean, (real) number or a value of the abstract set  $\mathbb{E}$  used for enumerations. The notation  $\llbracket a \rrbracket(v)$  denotes evaluating the expression  $a \in \text{Exp}(X)$  using valuation  $v \in V_X$ .

### EDA Semantics

The semantics of EDAs are based on a labeled transition system (LTS). The states of this LTS—here referred to as *configurations*—are pairs of modes and valuations. Transitions are subdivided in *discrete* and *timed* transitions. The prior are based on events, can change the current mode and update discrete values. The latter model the passing of time and update continuous variables.

The LTS that models an EDA  $\mathcal{A} = (M, m_0, X, v_0, \chi, \phi, E, \rightarrow, \Rightarrow)$  is defined as

$$\mathcal{L}_{\mathcal{A}} = (\text{Cnf}, \kappa_0, L, \longrightarrow),$$

where

- »  $\text{Cnf} = M \times V_X$  is the possibly infinite set of configurations;
- »  $\kappa_0 \in \text{Cnf}$  is the initial configuration;
- »  $L = \mathbb{R}_{>0} \cup E_\tau$  is the set of transition labels;
- »  $\longrightarrow \subseteq \text{Cnf} \times L \times \text{Cnf}$  is the transition relation.

The *temporal* modification of variables  $v + t \cdot \phi(m)$  denotes the change of the current valuation  $v \in V_X$  according to the passage of  $t \in \mathbb{R}_{>0}$  time units while the rates specified by  $\phi(m)$  are in effect. The update is defined as follows: For each local variable  $x \in LX$ , it holds that

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

The transition relation consists of *timed*- and *event* transitions (reactivation transitions are handled separately by the NEDA). It holds that  $(m, v) \xrightarrow{l} (m', v') \in \longrightarrow$  whenever:

- »  $l \in \mathbb{R}_{>0}$  and  $\chi(m)(v') = \top$  for  $t' \in (0, t]$  (meaning the mode constraint remains true), where  $v' := v + l \cdot \phi(m)$  and  $m' := m$ ;
- »  $(m, l, ex, g, f, m') \in \rightarrow$ , with  $v' := f(v)$ , when
  - $g(v) = \top$  (the guard is true) and
  - $\chi(m')(f(v)) = \top$  (the mode constraint of the target mode is true after applying the transition effects).

### Mapping Components to EDAs

For each component implementation, the local behavior can be described by an EDA with the above semantics. Summarizing, the following associations are used to make this mapping:

- » In and out data ports are mapped to input and output variables respectively;
- » Data subcomponents are mapped to local variables;
- » Event ports are mapped to events in the EDA. Furthermore, references to events of subcomponents in the SLIM components, such as `sub.e`, are mapped to events in the EDA of opposite direction;
- » Modes are identical between SLIM components and EDAs;
- » Mode invariants are mapped to mode constraints;
- » Trajectory equations are mapped to change rates;
- » Default values are mapped to the initial valuation;
- » Trigger expressions are mapped directly. In their absence, the empty expression  $\varepsilon$  is used.

The predicate  $\text{active}(e, m)$  is used to denote that the element  $e$  (contained in some component  $ci$ ), is active in mode  $m$ . This is true if  $e$  is specified either without an **in modes** specification, or with **in modes** ( $M$ ) where  $m \in M$ .

Formally, the mapping between component implementation  $ci$  and its associated component type  $ct$ , and its corresponding EDA

$$\mathcal{A}_{ci} = (M, m_0, X, v_0, \chi, \phi, E, \rightarrow, \Rightarrow),$$

is defined as follows:

- »  $M = \text{modes}(ci)$ ;
- »  $m_0 \in M$ , such that  $m_0$  maps to the unique **initial** or **activation** mode;
- »  $X = IX \cup OX \cup LX$  where
  - $IX = \{p \mid p \in \text{dataports}(ct) \wedge \text{dir}(p) = \text{in}\}$ ;
  - $OX = \{p \mid p \in \text{dataports}(ct) \wedge \text{dir}(p) = \text{out}\}$ ;
  - $LX = \text{datacomponents}(ci)$ ;
- »  $v_0 = \text{init}$ , with  $\text{init} \in V_X$ , denoting the *initial* valuation, which is determined by the *default* value assigned to output ports and data subcomponents, if any, and left undefined otherwise;
- »  $\chi(m) := \bigwedge_{i \in \text{invariants}(m)} \llbracket i \rrbracket$ ;
- »  $\phi(m)$  is defined as follows:

$$\phi(m)(x) := \begin{cases} r & \text{if } x' := r \in \text{trajectories}(m), \\ 1 & \text{if } x \in \text{clocks}(m), \\ 0 & \text{otherwise;} \end{cases}$$

- »  $E = IE \cup OE$ , where
- $IE = \{p \mid p \in \text{eventports}(ct) \cup \text{eventdataports}(ct) \wedge \text{dir}(p) = \text{in}\} \cup \{sc.p \mid sc \in \text{nondatacomponents}(ci), p \in \text{eventports}(sc.ct) \cup \text{eventdataports}(sc.ct) \wedge \text{dir}(p) = \text{out}\}$ ,
  - $OE = \{p \mid p \in \text{eventports}(ct) \cup \text{eventdataports}(ct) \wedge \text{dir}(p) = \text{out}\} \cup \{sc.p \mid sc \in \text{nondatacomponents}(ci), p \in \text{eventports}(sc.ct) \cup \text{eventdataports}(sc.ct) \wedge \text{dir}(p) = \text{in}\}$ .
- » For each transition  $m \text{ --}[t(tx) \textbf{ when } g \textbf{ then } e]\text{-->} m'$ , there exists a transition  $(m, e, ex, \llbracket g \rrbracket, f, m')$  in  $\rightarrow$ , where

- $e := \begin{cases} \tau & \text{if } t \text{ is the empty/internal trigger,} \\ sc.p & \text{if } t \text{ is of the form } sc.p, \text{ and} \\ p & \text{otherwise;} \end{cases}$
- $ex := \begin{cases} \perp & \text{if } tx \text{ is the empty expression } \varepsilon, \text{ and} \\ \llbracket tx \rrbracket & \text{otherwise;} \end{cases}$
- $\forall d \in IX. f(v)(d) := v(d)$ ;
- $\forall d \in OX. f(v)(d) := \begin{cases} \llbracket a \rrbracket(v) & \text{if } d := a \in e, \text{ and} \\ v(d) & \text{otherwise;} \end{cases}$
- $\forall d \in LX. f(d) := \begin{cases} \llbracket a \rrbracket(v) & \text{if } d := a \in e, \\ v_0(d) & \text{if } m_0 \text{ is an } \mathbf{activation} \text{ mode and} \\ & \neg \text{active}(d, m) \wedge \text{active}(d, m'), \text{ and} \\ v(d) & \text{otherwise.} \end{cases}$

- » Let  $\text{act} : M \rightarrow M$  be the function that maps a mode to its activation mode, where

$$\text{act}(m) := \begin{cases} m' & \text{if } m \text{ --}[\mathbf{@activation then } e]\text{-->} m' \text{ is a} \\ & \text{reactivation transition in } ci, \\ m_0 & \text{if } ci \text{ has an } \mathbf{activation} \text{ mode, and} \\ m & \text{otherwise.} \end{cases}$$

Then,  $\forall m \in M. (m, f, \text{act}(m)) \in \Rightarrow$ , with  $f$  is defined as follows:

$$f(v)(d) := \begin{cases} v(d) & \text{if } d \in IX, \\ \llbracket a \rrbracket(v) & \text{if } m \text{ --}[\mathbf{@activation then } e]\text{-->} \text{act}(m) \\ & \text{is a reactivation transition in } ci \text{ and } d := a \in e, \\ v_0(d) & \text{if } m_0 \text{ is an } \mathbf{activation} \text{ mode, and} \\ v(d) & \text{otherwise.} \end{cases}$$

**Example mapping** Using the above rules, the EDA of the GPS receiver implementation `gps_receiver.impl` shown in Listing 2.2 is defined as follows:



- »  $\mathcal{A}_i$  is the  $i$ 'th EDA in the model;
- »  $\alpha : M \rightarrow 2^{[n]}$  is the *activation mapping*;
- »  $EC : M \rightarrow (\{i.e \mid i \in [n], e \in E_i\} \times \{j.e \mid j \in [n], e \in E_j\})$  is the *event connection mapping*;
- »  $DD : M \rightarrow (\{i.x \mid i \in [n], x \in OX_i\} \rightarrow \{i.a \mid i \in [n], a \in \text{Exp}(IX_i \cup LX_i)\})$  is the *data dependence mapping*.

For a NEDA, the set  $M$  denotes the *global* set of modes, where  $M = \prod_{i \in [n]} M_i$ , with  $M_i$  being the set of modes for EDA  $i$ . Such global modes determine which components are active and what connections apply.

The activation mapping defines which EDAs are active given some current mode  $m \in M$ . That is, EDA  $i$  is active if and only if  $i \in \alpha(m)$ .

The event connection mapping defines which events are connected given the current mode, determining which transitions will synchronize. For a given mode, it specifies pairs  $(i.oe, j.ie)$  of some output event  $oe$  of EDA  $i$  and some input event  $ie$  of EDA  $j$  that are connected.

The data dependence mapping is a partial mapping that maps some expression over local or input variables to some output variable. For a given EDA  $i$ , if  $(j.x, i.a) \in DD(m)$ , then the value of port  $x$  of EDA  $j$  is determined by the expression  $a$  for EDA  $i$ . Note that unlike EC, DD is defined as a partial function since a data port cannot depend on multiple inputs, as its value is then not necessarily unique.

The first EDA in  $\{\mathcal{A}_i \mid i \in [n]\}$ , that is  $\mathcal{A}_1$ , is labeled the *root* EDA. In terms of a SLIM model, this is the root component that is at the top of the hierarchy (see also Section 5.2.1). An important consideration is that the ports of this component are not (explicitly) connected to anything. In particular, this means only ports of the form  $sc.p \in E_1$  can occur in EC.

### NEDA Semantics

Like for EDAs, the semantics of a NEDA are defined in terms of a labeled transition system, specifically as

$$\mathcal{L}_{\mathcal{N}} = (Cnf, \kappa_0, L, \Longrightarrow).$$

In the following,  $\mathcal{L}_{\mathcal{A}_i} = (Cnf_i, \kappa_{0,i}, L_i, \Longrightarrow_i)$  denotes the LTS associated with the constituent EDAs of  $\mathcal{N}$ .

Formally, the components of  $\mathcal{L}_{\mathcal{N}}$  are:

- »  $Cnf = \prod_{i \in [n]} Cnf_i$  the set of *global configurations*;
- »  $\kappa_0 = \langle \kappa_{0,1}, \dots, \kappa_{0,n} \rangle$  the *initial configuration*;
- »  $L = \mathbb{R}_{>0} \cup \{\tau\} \cup \bigcup_{i \in [n]} OE_i \cup IE_i$  the set of transition labels;
- »  $\Longrightarrow \subseteq (Cnf \times L \times Cnf)$  the transition relation.

Let  $\kappa = \langle (m_1, v_1), \dots, (m_n, v_n) \rangle \in Cnf$  be the current configuration, and  $\text{mod } \kappa = \langle m_1, \dots, m_n \rangle$  the currently active modes. The transition relation is constructed from *timed*, *internal* and *event* transitions as follows:

- » Timed transitions can be taken if all active components can take a timed transition of the same duration. Inactive EDAs will remain in the same configuration.

Thus,  $\forall t \in \mathbb{R}_{>0}. (\langle \kappa_1, \dots, \kappa_i, \dots, \kappa_n \rangle, t, \langle \kappa'_1, \dots, \kappa'_i, \dots, \kappa'_n \rangle) \in \implies$  if and only if  $\forall i \in \alpha(\text{mod } \kappa). (\kappa_i \xrightarrow{t} \kappa'_i) \in \implies_i$ .

- » Internal transitions do not communicate, and thus do not require the configuration of other EDAs to be taken into account.

Thus,  $(\langle \kappa_1, \dots, \kappa_i, \dots, \kappa_n \rangle, \tau, \text{nxt}(\alpha(\text{mod } \kappa), \langle \kappa_1, \dots, \kappa'_i, \dots, \kappa_n \rangle)) \in \implies$  if and only if  $\exists i \in \alpha(\text{mod } \kappa). (\kappa_i \xrightarrow{\tau} \kappa'_i) \in \implies_i$ .

- » Event transitions can only occur if all EDAs connected via EC can perform a transition synchronizing on the connected event. The origin is either an output event, or an input event from the root EDA,  $\mathcal{A}_1$ .

Thus,  $\forall e \in (\bigcup_{i \in [n]} OE_i) \cup IE_1. (\kappa, e, \text{nxt}(\alpha(\text{mod } \kappa), \kappa')) \in \implies$  if and only if

- $i \in \alpha(\text{mod } \kappa) \wedge (e \in IE_1 \vee (\kappa_i, e, \kappa'_i) \in \implies_i)$  and
- $\forall (i.e.j.e') \in EC(\text{mod } \kappa). (\kappa_j, e', \kappa'_j) \in \implies_j$ .

If a trigger expression  $ex$  is defined for the corresponding transition in EDA  $i$ , such that  $m_i \xrightarrow{e, ex, g, f}_i m'_i$  for some  $g$  and  $f$ , where  $\kappa_i = (m_i, v_i)$  and  $\kappa'_i = (m'_i, v'_i)$ , then  $v_i(e) := \llbracket ex \rrbracket(v_i)$ .

The target configuration  $\kappa' = \langle \kappa'_1, \dots, \kappa'_n \rangle$  is constructed by

$$\kappa'_k := \begin{cases} \kappa''_k & \text{if } e \notin IE_1 \wedge (\kappa_i, e, \kappa'_i) \in \implies_i, \\ \kappa''_k & \text{if } (i.e.k.e') \in EC(\text{mod } \kappa) \wedge (\kappa_k, e', \kappa'_k) \in \implies_j, \\ \kappa_k & \text{otherwise.} \end{cases}$$

In the definition of the transition relation above, the function  $\text{nxt} : (2^{[n]}, Cnf) \rightarrow Cnf$  makes a configuration consistent by updating the data dependence mapping, and the mode configuration depending on the activation mapping.

First the semantics for reactivation are defined. This is done recursively, as the activation of one component can trigger that of another. Note that for deactivation no update of the configuration is necessary, as it remains the same.

Let  $\beta(\kappa, \kappa')$  be the set of activated EDAs, i.e.  $\beta(\kappa, \kappa') = \alpha(\text{mod } \kappa') \setminus \alpha(\text{mod } \kappa)$ , and  $\text{activate}(\kappa, \kappa')$  the predicate that results in the configuration determined by applying the reactivation transitions for the EDAs specified by  $\beta$ , i.e.,

$$\text{activate}(\kappa = \langle (m_1, v_1), \dots, (m_n, v_n) \rangle, \kappa' = \langle (m'_1, v'_1), \dots, (m'_n, v'_n) \rangle) := \kappa'',$$

where  $\kappa'' = \langle (m_1'', v_1''), \dots, (m_n'', v_n'') \rangle$  with

$$(m_i'', v_i'') := \begin{cases} (m_i, v_i) & \text{if } i \notin \beta(\kappa, \kappa'), \\ (m_i''', v_i''') & \text{otherwise, where } m_i \xrightarrow{f} m_i''' \wedge f(v_i) = v_i'''. \end{cases}$$

Next, the effect of the data dependence mapping on the configuration is defined. Valuations are updated according to existing connections, or when a connection is broken, leading to the value being reset to the initial value.

Let  $\text{cns}(\kappa, \kappa')$  result in the configuration determined by applying the effects of the data dependence mapping, i.e.

$$\text{cns}(\kappa, \kappa' = \langle (m_1', v_1'), \dots, (m_n', v_n') \rangle) := \kappa'',$$

with  $\kappa'' = \langle (m_1'', v_1''), \dots, (m_n'', v_n'') \rangle$  and

$$\begin{aligned} \forall i \in [n], d \in IX_i \cup OX_i. \\ v_i''(d) := \begin{cases} \llbracket a \rrbracket(v_i') & \text{if } \exists j \in [n]. \text{DD}(\langle m_1', \dots, m_n' \rangle)(i.d) = (j.a), \\ v_{0,i}(d) & \text{else if } \exists j \in [n]. \text{DD}(\text{mod } \kappa)(i.d) = (j.a), \\ v_i'(d) & \text{otherwise.} \end{cases} \end{aligned}$$

The next configuration is then determined as

$$\text{nxt}(\kappa, \kappa') := \text{cns}(\kappa', \text{activate}(\kappa, \kappa')).$$

### *Mapping a SLIM Model to a NEDA*

A complete SLIM model can be mapped to a NEDA, where each individual component is represented by an EDA, but connections between components, as well as their (de)activation are governed by the NEDA.

First, each component in the hierarchy of the SLIM model is mapped to its corresponding EDA as described in Section 2.5.1. The root component is always mapped to  $\mathcal{A}_1$ , subsequent components to  $\mathcal{A}_2, \dots, \mathcal{A}_n$ ,  $n$  being the number of components and corresponding to  $[n]$ .

The activation mapping  $\alpha$  is constructed such that the root component is always active, and subcomponents can only be active when their parent is. More specifically,

$$\forall i \in [n]. i \in \alpha(\langle m_1, \dots, m_n \rangle) \iff i=1 \vee \exists j. \text{active}(i, m_j) \wedge j \in \alpha(\langle m_1, \dots, m_n \rangle).$$

The event connection mapping is constructed by taking the transitive closure of connections active in a given global mode. First, the direct connection relation  $\text{EC}^1$

is defined as follows:

$$\begin{aligned} EC^1(M = \langle m_1, \dots, m_n \rangle) := & \\ & \{(j.p, i.j.p) \mid i, j \in [n] \wedge i \neq j \wedge j \in \text{subcomponents}(i) \wedge j \in \alpha(M) \wedge p \in OE_j\} \cup \\ & \{(i.j.p, j.p) \mid i, j \in [n] \wedge i \neq j \wedge j \in \text{subcomponents}(i) \wedge j \in \alpha(M) \wedge p \in IE_j\} \cup \\ & \{(sc_1.p, sc_2.q) \mid c = \mathbf{port} \ sc_1.p \rightarrow sc_2.q \in \text{connections}(i) \wedge \\ & sc_1, sc_2 \in (\text{subcomponents}(i) \cup \{\varepsilon\}) \wedge \text{active}(c, m_i)\}. \end{aligned}$$

Then, EC is the transitive closure of that, such that

$$(p, q) \in EC(M) \iff (p, q) \in EC^1(M) \vee \exists r. (p, r) \in EC(M) \wedge (r, q) \in EC(M).$$

The data dependence mapping is generated from data flows and connections such that  $\forall i \in [n], x \in (IX_i \cup OX_i)$ :

$$DD(\langle m_1, \dots, m_n \rangle)(i.x) := \begin{cases} \llbracket a \rrbracket & \text{if } (\mathbf{flow} \ a \rightarrow i.x = f) \in \text{flows}(i) \wedge \\ & \text{active}(f, m_i), \text{ and} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Here,  $\mathbf{port} \ p \rightarrow q$  is considered an alias for  $\mathbf{flow} \ p \rightarrow q$  if  $p$  and  $q$  refer to data ports.

**Example mapping** As an example, the system from Listing 2.3 is mapped to a NEDA using the principles above. As there are three components in total, it holds that  $n = 3$ . The EDAs are assumed to exist for all the systems, with  $\mathcal{A}_1$  being the root EDA for the `gps.impl` system, and  $\mathcal{A}_2$  and  $\mathcal{A}_3$  the EDAs for `gps_receiver.impl` as defined in Section 2.5.1. The modes and activation mapping are constructed in a straightforward manner:

$$\begin{aligned} M = & \{m_{\text{none}_1}, m_{\text{prim}_1}, m_{\text{sec}_1}\} \times \{init_2, acquire_2, enabled_2, failed_2\} \times \\ & \{init_3, acquire_3, enabled_3, failed_3\} \\ \alpha = & \{m \mapsto \{\mathcal{A}_i \mid i \in [n]\} \mid m \in M\}. \end{aligned}$$

The connections between the root component and its subcomponents are always present and do not depend on the mode. These are defined as

$$\begin{aligned} EC_{\text{sub}} = & \{ (2.has\_fix, 1.primary.has\_fix), (2.lost\_fix, 1.primary.lost\_fix), \\ & (2.position, 1.primary.position), (3.has\_fix, 1.secondary.has\_fix), \\ & (3.lost\_fix, 1.secondary.lost\_fix), (3.position, 1.secondary.position) \}. \end{aligned}$$

The event connections and data dependence relations can then be constructed using these definitions. Note that they depend only on the modes of the root component, those of the subcomponents do not affect them. Making use of this fact,

the following definitions suffice:

$$\begin{aligned}
EC(\{m \mid m \in M \wedge m_1 = m\_none_1\}) &:= EC_{\text{sub}} \\
EC(\{m \mid m \in M \wedge m_1 = m\_prim_1\}) &:= EC_{\text{sub}} \cup \{(2.\text{position}, 1.\text{position})\} \\
EC(\{m \mid m \in M \wedge m_1 = m\_sec_1\}) &:= EC_{\text{sub}} \cup \{(3.\text{position}, 1.\text{position})\} \\
DD(\{m \mid m \in M \wedge m_1 = m\_none_1\}) &:= \{(\perp \mapsto 1.\text{fix})\} \\
DD(\{m \mid m \in M \wedge m_1 = m\_prim_1\}) &:= \{(\top \mapsto 1.\text{fix})\} \\
DD(\{m \mid m \in M \wedge m_1 = m\_sec_1\}) &:= \{(\top \mapsto 1.\text{fix})\} .
\end{aligned}$$

### 2.5.3 NON-DETERMINISM

If more than one transition is possible — i.e., enabled — the choice on what transition to take is made *non-deterministically*. This effectively means that any analysis will consider all possibilities. In particular, for a NEDA, in any given configuration  $\kappa$ , a possibly infinite set of successor configurations  $\kappa'$  can be reached in a single step via transitions  $\kappa \xrightarrow{l} \kappa'$ , with  $l \in L$ .

Variables for which the valuation is undefined provide a further source of non-determinism. Such variables hold an arbitrary value from their domain, that is not specified anywhere in the model. This is particularly relevant for the variables in  $IX_1$ , which are the ports on the root component. These port can take any value at any point in time, which is part of what is referred to as the *open world assumption*, the other part being that transitions triggered by  $IE_1$  are always possible as well. Furthermore, *uninterpreted functions* can be specified by the model, which given some input value, result in an arbitrary, but fixed, output value.

### 2.5.4 PROBABILISTIC SEMANTICS

Aside from the hybrid semantics described above, a probabilistic interpretation of the NEDA can be used as well, which is based on event *rates*, which in SLIM are defined using error rates on error events. In the case of probabilistic systems, or rather models that contain probabilistic aspects, properties reason over the probabilistic *measure* of possible traces in the model. In the case of SLIM and its NEDA, these measures are based on continuous distributions.

First, the transitions and transition rates of probabilistic events have to be considered. If in SLIM an event is defined with **occurrence poisson**  $\lambda$ , it is associated with the *exponential* distribution with rate  $\lambda$ . This means that given a time bound  $t$ , the probability of this event occurring within  $t$ , assuming the transition is enabled, is  $1 - e^{-\lambda t}$ .

If the model is non-deterministic, a scheduler is required to resolve this. In general, such schedulers can either be minimizing or maximizing for the sought-after probability. Such schedulers function by mapping the possible, non-deterministic

paths to a probability distribution (in such a way that they are *measurable*). This will allow the model to be interpreted in a fully probabilistic fashion again, making it possible to determine the probability of paths in the model.

### Interactive Markov Chains

The fully probabilistic (i.e., untimed) underlying model corresponds to an interactive Markov chain (IMC) [79], which can be defined as a tuple

$$\mathcal{I} = (S, L_a, \rightarrow, \Rightarrow, s_0),$$

of which

- »  $S$  is the finite set of states;
- »  $L_a$  is the finite set of actions, including internal action  $\tau$ ;
- »  $\rightarrow \subseteq S \times L \times S$  is the action transition relation;
- »  $\Rightarrow \subseteq S \times \mathbb{R}_{>0} \times S$  is the Markovian (or probabilistic) transition relation;
- »  $s_0$  is the initial state.

The function  $\text{rate} : OE \rightarrow \mathbb{R}_{\geq 0}$  results in the rate associated with some output event, and is defined as follows:

$$\text{rate}(e) := \begin{cases} r & \text{if } e \text{ was originally defined with } \mathbf{occurrence\ poiss}\ \mathbf{r}, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

The construction of such an IMC is based on the LTS of the NEDA. The states of the IMC are based on the configurations of the NEDA. The action transitions are based on those transitions in the NEDA that are neither timed, nor associated with a rate. The Markovian transitions are based on the transitions that are associated with a rate.

Given an NEDA  $\mathcal{L}_N = (Cnf, \kappa_0, L, \Longrightarrow)$ ,  $\mathcal{I}$  is defined as follows:

- »  $S := Cnf$ ;
- »  $L_a := L \setminus \mathbb{R}$ ;
- »  $\rightarrow := \{(\kappa, l, \kappa') \mid (\kappa, l, \kappa') \in \Longrightarrow \wedge l \notin \mathbb{R} \wedge (l \notin OE \vee \text{rate}(e) = 0)\}$
- »  $\Rightarrow := \{(\kappa, \text{rate}(e), \kappa') \mid (\kappa, l, \kappa') \in \Longrightarrow \wedge l \in OE \wedge \text{rate}(e) > 0\}$
- »  $s_0 := \kappa_0$ .

**Maximal Progress** In IMCs, action transitions are considered to be *instantaneous*, whereas Markovian transitions are taken after some delay governed by its rate (or rather, the probability distribution that depends on it). This observation leads to the following: If there exists  $s, s', s'' \in S, l \in L, \lambda \in \mathbb{R}_{>0}$  such that  $(s, l, s') \in \rightarrow \wedge (s, \lambda, s'') \in \Rightarrow$ , then the Markovian transition  $(s, \lambda, s'')$  will be ignored. The reasoning is that the probability of taking this transition instantaneously, that is with zero delay, is zero, as  $1 - e^{-\lambda t} = 0$  if  $t = 0$ . This is called the *maximal progress assumption* [80].

### 2.5.5 INVALID BEHAVIORS

It is possible for the NEDA described by the model to exhibit behavior which is either unwanted or simply invalid. Such behavior can occur as either it is possible to syntactically restrict it, or doing so would be undesirable as it would limit the expressivity of the language too much. In the following, such behavior is classified.

Perhaps the best known example is the *deadlock*, which is a configuration  $\kappa$  from which no other configuration is reachable, i.e.,  $\nexists \kappa' \in \text{Cnf}, l \in L. \kappa \xrightarrow{l} \kappa'$ . Under the semantics of SLIM, such a configuration is considered invalid. COMPASS offers the functionality to check if such states are reachable for discrete systems, see Section 5.3.3 (where the ramifications of their existence is discussed as well), but this check is not available for timed/hybrid systems.

For timed systems, two different invalid behaviors may occur: *Zeno cycles* and *time-divergence*. First, Zeno cycles are traces of infinite length where the total sum of elapsed time is finite. This means that an infinite number of discrete steps have occurred over a finite amount of time, which is unrealistic and affects some analysis, as will be discussed in Section 4.3.7.

Time divergence is the occurrence of traces where the value of one or more clocks grows unbounded (i.e., clocks that are never reset). Though the concept in and of itself may be expected, it is nevertheless not allowed for SLIM, as it has a negative influence on analysis similar to deadlocks. Again, this is further discussed in Section 5.3.3.

Finally, the invariants specified for the model must be convex at all times. This means that for any reachable state, there may be no delays  $d_1, d_2, d_3 \in \mathbb{R}_{\geq 0}$  with  $d_1 < d_2 < d_3$ , such that an invariant of the active modes is true at  $d_1$  and  $d_3$ , but false at  $d_2$ .

## 2.6 DIFFERENCES BETWEEN SLIM AND AADL

Although SLIM is based on AADL, there are some significant differences. On the one hand, SLIM features the subset of AADL most critical to validation and verification of aerospace systems within the scope of COMPASS. On the other hand, some extensions were made, again for the purpose of V&V. Therefore, SLIM can be described as an “extended subset” of AADL. Recent efforts have closed the gap between the two to some degree, but differences remain. The following gives a quick summary of the differences.

**Error Specifications** In AADL, error modeling is possible by means of the error modeling annex (EMA)—which at the time of writing is specified in [AS5506-1A]. As the name implies, this is an annex for AADL, meaning it is not directly embedded in the language, but available as an extension. In the EMA, error specifications are modeled directly as part of a component, as opposed to SLIM where they are

provided as separate entities. This directly affects how they are integrated, or bound, into the system. In the EMA, errors can propagate along existing ports in the nominal model, or other connection points. In SLIM, a separate, implicit, connection is provided for this. Furthermore, the EMA provides a separate type hierarchy for possible errors. SLIM does not have this notion, where errors occur as untyped events or propagations. Finally, SLIM permits the error model to directly influence the data and behavior of the model by means of fault injections. As the behavioral aspects of AADL are covered in a separate annex, the behavioral annex, such a link is not possible with the EMA. The EMA, however, can influence the events that occur between components and, thanks to its type system, provides more fine grained control on the flow and transformation of errors.

**Behavioral Specifications** A major difference between AADL and SLIM remains the specification of behavior. For AADL, the behavioral annex [AS5506-2] permits the definition of state machines as part of a component. Syntactically these are similar to the states specified by SLIM, though already at first glance there are some key differences:

- » The state machine functions more like a program, with an initial and return state. In fact, they are often exemplified as the behavior of a subprogram component, a category that SLIM does not support.
- » Data values are passed as component *parameters*, whereas SLIM defines these using data ports. Local variables are provided as *state variables* in the annex, whereas SLIM uses data subcomponents.

The main difference regarding behavioral and error specifications with respect to AADL remains, however, that in SLIM, they are directly integrated into the language, whereas for AADL they are specified by means of annexes. This permits SLIM to consider the combination of both aspects directly, instead of individually. Furthermore, the formal aspects of SLIM can be fully specified this way.

**Component Execution** In SLIM, all components are executing concurrently, performing either internal transitions or synchronizing event transitions. In AADL, concurrency is modeled solely using **thread** components, of which the execution is based on the dispatch policy specified in the model (AADL defines properties for this), and can be further controlled by receiving events which may trigger the dispatch of a thread.

**Port Connections** The semantics of ports and port connections differ between AADL and SLIM as described in Section 2.2.1. In particular the **flow** style of connection is a concept only known to SLIM. Furthermore, events in SLIM are synchronous, requiring transitions to synchronize on them. In AADL, events and data are *sampled* (though with various possible configurations), depending on the dispatch rates of the involved components.

**Properties** AADL defines a set of properties that differs from that of SLIM. For the most part, the AADL properties pertain only to the subset that is not supported by SLIM, and hence are left out. SLIM in turn defines some properties that are relevant to describing the behavior of certain aspects of the model that are not relevant to AADL.

## 2.7 FUTURE WORK

Although SLIM is already a practically applicable modeling language, further improvements can still be made. One point remains the current differences between SLIM and AADL. With SLIM 3.0 efforts have been made to close this gap, but further differences still exist that could be consolidated.

Of particular interest are the behavioral and error annexes that have been defined for AADL. The SLIM behavioral and error specifications have been (loosely) based on past versions of these annexes, but newer versions exist. Updating SLIM to be closer in line with these would make it easier to adopt for those who are already familiar with them, as well as open up possibilities to use AADL-based tooling for SLIM models, and vice versa.

### 2.7.1 AADL v3

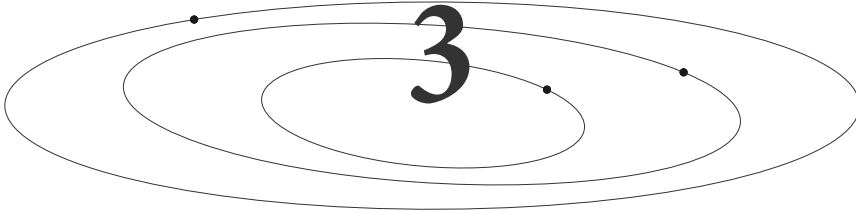
AADL is still being developed and maintained, which also includes the addition of new features and extensions of existing ones. The latest version of AADL is v2.2 [AS5506C], with the next version of AADL (v3) being in the design stages. AADL v3 introduces some new concepts that should be considered for adoption in the SLIM language.

First and foremost, a unified type system is considered. In this unified type system, the same types will be used for data components as well as properties, and introduce a few basic types for Booleans, integers, real numbers and strings. Adopting such a unification removes some of the disparity between SLIM and AADL, as SLIM can then adopt AADL data types instead of defining its own (built-in). This also applies to types defined in SLIM that cannot directly be mapped to the basic AADL types, by means of user-defined types such as can be found currently in property sets. This may include things such as bounded integers, ranges, or types with some attached unit (such as time).

Expressions will also be supported beyond those that currently exist for Boolean properties. Currently, SLIM requires expressions to be provided by means of a property string, as no specialized parser rules have been provided (as is the case for default values for example). Having expressions be part of the AADL language also permits this solution to be replaced with a first-class language construct, improving usability.

Note that at the time of writing, no concrete syntax for these features exists, nor any guarantee that they will be adopted in the next major release of AADL. Nevertheless,

these features will be helpful to provide a closer match between SLIM and AADL, ultimately making it easier to adopt SLIM for someone that is already familiar with AADL.



## SPECIFYING REQUIREMENTS

*ABSTRACT – This chapter discusses the specification of requirements in the scope of formal verification of systems. It starts by defining a taxonomy of requirement types commonly found in aerospace projects. With regard to the formalization process, automation of requirement formalization—or even of the specification of the requirements—is a key point. Techniques using patterns or design attributes are treated. The prior use placeholders in sentences, formulated using natural language. The latter make use of elements in the model, and assign some values to them which match those specified by the requirements. On a formal level, the requirements are translated into formulas using a variety of logics. These formulas can then be verified against the formal semantics of the model.*



**E**FFECTIVELY any project's success depends on the correct specification of its requirements. They define the scope and goals of the project and, as they get progressively more detailed, constraints and guidelines [93, 117]. Hence, a lot of effort is spent on tracking requirements, tracing them to the various artifacts produced during the project's development, as well as validating and verifying them.

It is a well-known fact that any defect in the requirements, it being a contradiction, oversight or otherwise, can lead to costly consequences when detected only in the later stages of the project. Therefore, being able to validate and verify these requirements early on is highly beneficial [96, 125]. However, as most requirements are specified rather informally, usually in plain natural language, it is hard to do so consistently. Using formal methods enables a much more structured approach, but this requires a formal representation of the requirements. In this chapter, specifically Section 3.3, approaches to tackle this problem are discussed. It also gives a more detailed description of the approach taken by the COMPASS toolset. The aforementioned refinement of requirements is also discussed. As a project progresses,

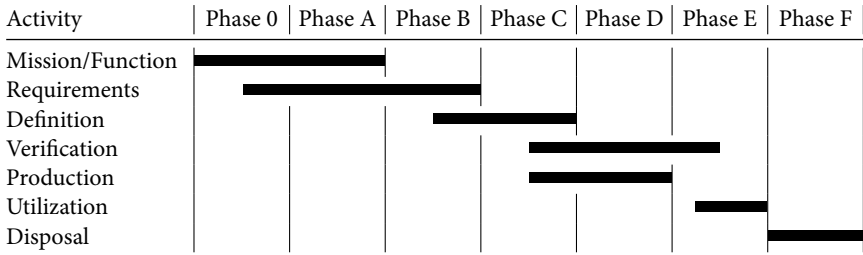


FIGURE 3.1 – Overview of project lifecycle. This lifecycle is defined by [MST10C], which also provides more details about the phases and activities.

abstract requirements are (gradually) made more concrete. In this process, it is important to ensure that they remain consistent, and the satisfaction of requirements is preserved under refinement.

### 3.1 REQUIREMENTS IN AEROSPACE

During several phases of the space system project lifecycle, requirements are defined and refined. This means that the scope of requirements and their level of detail changes as the project progresses.

The various phases of a project lifecycle are well established, with the requirements being treated in the early phases as shown in Figure 3.1. A project starts in phase 0—Mission analysis—and continues through Phases A–F. Requirement engineering starts in phase 0, followed by phases A and B—Feasibility and Preliminary definition, respectively. Later phases involve the implementation and execution of the project.

Two import aspects play a role when specifying requirements during the various phases: the types of requirements that are specified and the refinement of requirements specified in earlier phases. For the prior a taxonomy can be constructed, which is described in the following section. The latter is addressed in Section 3.3.3.

#### 3.1.1 ABSTRACTION LEVELS

During the development of space projects, various levels of abstraction are traversed as the project progresses. At the starting point, only the highest levels of abstraction are considered, and are refined further to lower levels towards the end. The various levels of abstraction are related through *realization* or *refinement*, where a lower level of abstraction realizes or refines a higher level one. This in turn means that project items at higher abstraction levels are described in more detail by those of lower levels.

The abstraction levels considered in this chapter are based on those of [EHB1002A] as they apply specifically to the engineering of aerospace systems. They can be

described as follows:

1. **Mission:** At this level the entire mission is described, with its goals and expected products and services.
2. **Overall-system, or System-of-systems:** This level considers the system solution to achieve the goals specified by the mission. This encompasses such entities as the ground, launch, space, and support segments.
3. **System:** At this level the various elements of the overall-system level items are considered individually, such as launchers and satellites.
4. **Sub-system:** The sub-system level considers the components out of which system level items consist. For example, a satellite contains control for avionics and attitude, power distribution and software systems.
5. **Equipment:** The lowest considered level, for which a further level of refinement is out of scope. Example elements are for instance batteries, processors and sensors.

Requirements can be specified at the various abstraction levels, and be further refined by those specified at lower abstraction levels (which will be further discussed in Section 3.3.3). Moreover, certain classes of requirements are only introduced at specific levels in the abstraction hierarchy, and refined further down (insofar applicable). Therefore, when classifying requirements, as discussed in the next section, the abstraction level should be considered as well.

Concerning the formalization of requirements within COMPASS, only the system, sub-system, and equipment levels are considered, as these can be represented by the SLIM modeling language. Requirements specified at higher levels of abstraction can be considered and linked into the model by means of the mission specification as discussed in Chapter 5, but will not be further considered in the remainder of this chapter.

## 3.2 CLASSIFYING REQUIREMENTS

Distinguishing between various types of requirements is a well-established practice, most notably the separation between functional and non-functional requirements, and the further subdivisions thereof, e.g., see [42]. This section presents a taxonomy of requirement types that are typically found in the space domain. Note however, that throughout the various industries and disciplines there is not always agreement on how classifications should be made [64]. However, within the scope of the COMPASS project, the primary target domain is that of space projects, for which standards exist that can be used for this purpose. Therefore, the taxonomy is largely based on [EST1006C].

An overview of the taxonomy is given in Figure 3.2. At the top level, requirements are categorized as technical requirements and process requirements. The technical

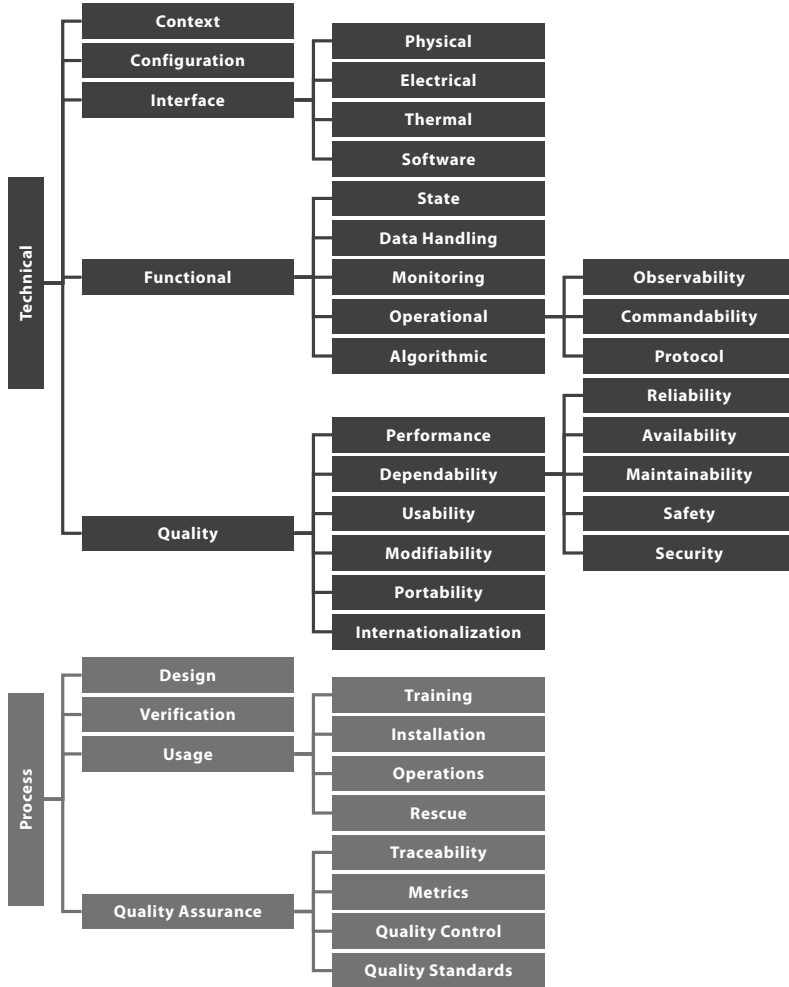


FIGURE 3.2 – Overview of the requirements taxonomy.

requirements define the “*required technical capability of the product in terms of performances, interfaces and operations*” [EST1006C], whereas process requirements defines the required processes to be followed by people to develop, use and maintain the system. Process requirements are generally not formalizable, and therefore not considered for formalization. The technical requirement classes are considered on a case-by-case basis.

Applying the taxonomy to requirements is not a simple one-to-one mapping, but may involve having multiple classes be applicable to a single requirement. This is particularly true for more complex requirements, though this is an indication the requirement itself may be better specified as two separate requirements.

The taxonomy presented here is not unique in its nature, and other taxonomies exist, also in the aerospace domain. Rather, it has been designed with the formalization of requirements in mind, in particular geared towards use in the COMPASS toolset.

### *Technical Requirements*

Technical requirements provide specifications of the system under development. They describe characteristics of the system, but not, e.g., how it is developed or used.

From a formal point of view, these requirements are the most interesting, as they describe the system itself; in particular what functions are provided, how they are to be implemented and under which constraints. See Section 3.3 for more details. The classes defined in the technical category are listed below:

1. **Context Requirements:** These requirements specify the context of the system. Such a context can consist of external devices, physical conditions and resources. This class of requirements encompasses the *Environmental* and *Physical* requirement classes of [EST1006C].
2. **Configuration Requirements:** These requirements specify the composition of a system, both sub-systems and their connections. As such, these requirements provide a direct link to requirements of lower abstraction levels.
3. **Interface Requirements:** These requirements specify both physical and virtual interfaces of the system. Further subclasses are defined as follows:
  - a) **Physical Interface Requirements:** Requirements concerning the physical constraints regarding various physical dimensions, such as geometry, mass and inertia.
  - b) **Electrical Interface Requirements:** Specification of electrical properties of a system. This also includes margins on the electrical effects the system may have on its (nearby) environment.
  - c) **Thermal Interface Requirements:** Specify properties such as conductivity (thermal resistance).
  - d) **Software Interface Requirements:** These specify the interfaces which software may use to interact with other software or hardware.

4. Functional Requirements: Specify what the system should do. This class of requirements also entails the *Mission* requirements class of [EST1006C], which are considered functional requirements at the mission abstraction level. These requirements are of a qualitative nature. A set of subclasses is defined as follows:
  - a) Operational Requirements: These requirements specify the rules according to which the system can communicate. Such rules include the commands and results that may be applicable, as well as the sequences in which they occur.
    - i. State Requirements: These requirements describe the possible states and state transitions of the system. States define how the system is to respond to inputs, transitions can specify conditions and triggers relating to their behavior. State requirements also encompass the specification of state invariants.
    - ii. Protocol Requirements: They specify the order and format of messages used for system communication.
  - b) Algorithmic Requirements: Specify the relation between in- and output in a functional manner, as, e.g., formulas, (pseudo-)code or textual description.
5. Quality Requirements: Specify the way a system must perform and the characteristics it must have for it to allow developers, maintainers and users to perform their tasks related to the system.
  - a) Performance Requirements: Specify the performance of the system according to various metrics, e.g., latency or processing capacity.
  - b) Dependability Requirements: The specifications that relate to the dependability of the system, such as possible failures and their effects. Dependability requirements can be further subdivided according to the RAMS attributes:
  - c) Reliability Requirements: Specify reliability metrics, such as mean time between failures (MTBF).
  - d) Availability Requirements: Specify availability metrics. Availability can be considered the fraction of time, on average, the system is available.
  - e) Maintainability Requirements: Specify the acceptable amount of effort required to maintain the system in an operational status, or bring it back to it.
  - f) Safety: Specify possible system hazards (internal and external) and the amount of associated risk that is acceptable.
6. Modifiability Requirements: Specify the effort required to adapt the system to new applications.
7. Portability Requirements: Specify the effort required to adapt the system to new environments.

8. Internationalization Requirements: Specify the effort required to adapt the system to new (natural) languages.

### *Process Requirements*

Process requirements specify the tasks to be performed by people in order to develop, maintain and use the system. Such requirements identify the roles of users, as well as their associated tasks. Process requirements also identify tools, input and outputs for associated activities.

In the following, the various subcategories of process requirements are described. Note, however, that in the rest of this chapter, such requirements will no further be referenced. This is because of the fact that these requirements do not describe the system itself, and are not suitable for the types of formal analysis discussed in the rest of this chapter.

1. Design Requirements: Such requirements are used to ensure that the design of the system is performed in a systematic and organized way.
2. Verification Requirements: These requirements specify how the system is to be verified, in particular with respect to other specified requirements.
3. Usage Requirements: Requirements that specify how the system should be operated by users, in particular which processes are to be followed. They can be further subdivided as follows:
  - a) Operational Use Requirements: Specify the processes that operators have to follow in order to achieve the objectives of the system.
  - b) Operational Maintenance Requirements: Specify the processes that are to be followed for maintenance of the system.
  - c) Handling Requirements: Specify the processes that are to be followed for handling the system.
  - d) Training Requirements: Specify the processes that are to be followed in order to train users on using the system.
  - e) Installation Requirements: Specify the processes that are to be followed to install the system.
4. Quality Assurance Requirements: Specify how processes are executed correctly.

## 3.3 FORMAL ANALYSIS

Considering the importance of technical requirements in the design process, there is a high demand for validating and verifying them. The earlier this can be achieved, the more cost-efficient any required changes may be applied. *Formally* verifying them provides the best possible guarantees, as correctness can be mathematically proven.

A variety of methodologies exist around formalization targeting the various aspects of requirements engineering such as their management, elicitation, representation, validation and verification. This section looks into the formalization of requirements, in particular with respect to the approach taken in COMPASS, as well as formally representing and validating the refinement of requirements.

One major benefit of a formal approach is that the impact of changing a requirement can much easier be estimated. Updating the formal representation of the requirement allows subsequent analysis to be performed automatically. Furthermore, any artifacts derived from these specifications can be updated as well.

### 3.3.1 FORMALIZATION

The process of formalizing requirements is a long considered problem, with a variety of approaches. At the basis, some formalism is required to represent the requirement in the form of a (formal) property. Different formalisms provide varying levels of expressiveness, which is often correlated to the complexity of algorithms that can be used for analysis. An overview will directly follow.

Aside from the formalism chosen, the formalization step itself can also be treated in various ways. Directly formalizing a requirement by hand requires a great deal of expertise, and is error prone. Automating this task alleviates this. Although full automation is the ideal, some degree of manual translation is often required.

#### *Formalisms*

Perhaps the best known formalisms to represent properties (derived from requirements) are temporal logics. Typical examples are linear temporal logic (LTL) and computation tree logic (CTL) [49, 116], which apply to discrete systems. These logics capture behavior along paths or trees respectively, and permit the construction of complex specifications. In Appendix A, these and other logics are described in more detail, including their formal definition.

For real time systems, metric temporal logic (MTL) [94] is an applicable logic. MTL allows reasoning over continuous time intervals on the ‘until’ operator, making it suitable for the specification of timed properties. A subset of MTL is metric interval temporal logic (MITL) [6], which prohibits point intervals.

On the probabilistic side, well-known logics are probabilistic computation tree logic (PCTL) [73] and continuous stochastic logic (CSL) [12, 13]. Whereas PCTL lends itself well for discrete probabilistic systems, it is currently not known whether its associated satisfiability problem is decidable. CSL is well suited for the continuous domain. The latter is used by COMPASS, as SLIM specifies probabilities by means of error rates.

Other approaches make use of automata to formally represent system properties, such as Büchi, Muller and Rabin automata [66], or timed automata [39]. Other

formalisms used include abstract state machines (ASMs) [72], Z [129], B [2], Alloy [82] and OCL [OCL]. However, these will not be discussed further as they are not used within the framework of COMPASS.

**Safety and liveness** One aspect of temporal properties is that specific instances can be categorized as *safety* or *liveness* properties [4, 116]. Intuitively, safety properties are satisfied when something bad does not happen, whereas liveness properties specify that some good behavior occurs eventually. Safety and liveness properties have been characterized for the linear-time [5, 88] and branching-time settings [28, 97], as well as in the probabilistic setting [86]. Furthermore, in the linear-time setting, every LTL property can be written as a conjunction of a safety and liveness property.

The ability to distinguish between the two fragments is useful for verification. Different algorithms are applicable, or even required, for either category. A specialized algorithm may be more efficient than one that handles all instances of a property language. For instance, safety properties only need to consider a finite path (prefix). Highlighting the difference, model checking or checking satisfiability of the safety fragment of MTL [114] is decidable, but is undecidable in general.

#### *Pattern-based Formalization*

So far the discussion considered approaches to represent requirements, though not the methodology to perform the actual formalization. A well studied approach to formalization, and one that is used by COMPASS, is the use of patterns [55]. The concept of patterns is that many requirements are phrased in often recurring ways, allowing patterns to be derived from them. Based on known requirements a set of patterns can be built, each pattern containing one or more *placeholders*, which may be replaced with concrete statements about the system. To use these patterns, a user (domain expert) would map a requirement to a patterns, and fill in the placeholders based on the details of the requirement. Each pattern is mapped to a formula a priori, with the placeholders representing (usually atomic) propositions.

A variety of catalogs exist that define a set of patterns for a particular class of systems, e.g., discrete or timed, as well as those that combine them. Initially, the work of [55] defined a number of patterns for qualitative properties, including LTL and CTL. Later, in [16, 92] patterns for real-time systems have been introduced. Finally, [68] covers a set of patterns for probabilistic systems. Patterns from these works generally apply to various systems, and can be applied to the aerospace domain. Other, more specific approaches exist as well, such as for security [40] and epistemic properties [33].

Between all these approaches, a common set of patterns can be found, which is used as a basis for the work in [10]. COMPASS makes use of a pattern-based approach as well, using this set as the basis. In particular, this set supports qualitative, timed and probabilistic properties, which coincides with the semantics supported by the SLIM language.

Patterns are constructed based on two factors: Their *class*, and their *scope*. The pattern classes define the overall structure of the pattern, and determine when the embedded proposition(s) should hold. More specifically the classes relate to the materialization of propositions, e.g., whether they appear once, always or never, and the relation with other propositions, e.g., whether they occur before or after some other proposition. The scope determines during which period(s) the property specified by the class should hold.

Classes are primarily divided into groups, namely *occurrence* and *order*. The occurrence classes describe the expected lifetime of single properties, whereas order classes the relation of the lifetime between different properties. Both groups are summarized below.

As mentioned, classes are instantiated with propositions. In the descriptions below, these are indicated by  $\varphi$  and  $\psi$ . Note that the example formulas are just indicative, and vary between the various logics that a pattern may be mapped to.

### Occurrence classes

- » Existence: The proposition  $\varphi$  shall eventually hold. Generally mapped to the *exists* operator (i.e.,  $\diamond\varphi$ ), this class of pattern is satisfied when  $\varphi$  holds true at least once.
- » Universality: The proposition  $\varphi$  always holds. This maps to the *always* operator (i.e.,  $\square\varphi$ ). This class is satisfied when the proposition holds true without interruption.
- » Absence: The opposite of universality. This class requires the proposition to never hold true (i.e.,  $\square\neg\varphi$ ).
- » Recurrence: the proposition  $\varphi$  shall eventually hold, regardless whether it held previously or not. This can be considered a combination of both the always and existence operators (i.e.,  $\square\diamond\varphi$ ).

### Order classes

- » Precedence: The proposition  $\varphi$  is always preceded by the occurrence of proposition  $\psi$ , i.e., some other proposition must have held in the past.
- » Response: The proposition  $\varphi$  is always followed by the proposition  $\psi$ . Similar to precedence, this class considers another proposition to happen in the future.
- » Response-Invariance: Once the proposition  $\varphi$  holds, the proposition  $\psi$  holds continuously.
- » Until: The proposition  $\varphi$  holds until proposition  $\psi$  occurs (i.e.,  $\varphi\mathcal{U}\psi$ ). Important to note is that  $\psi$  is required to occur for the pattern to be satisfied.

The scope of patterns determine during which period in the lifetime of the whole system the specification of the class should hold. Like the classes, scopes can be

parameterized. Here, these parameters will use the variables  $\rho$  and  $\omega$ . The following scopes are defined:

- » Global: The whole lifetime of the system is considered. In other words, the lifetime considered is not restricted.
- » Before: The pattern is only considered before some other proposition  $\rho$  holds.
- » After: The pattern is only considered after some other proposition  $\rho$  holds.
- » Between: The pattern is only considered after some proposition  $\rho$ , up to some proposition  $\omega$  holding true. In this case,  $\omega$  is not required to occur (weak until).
- » After-Until: Like the between scope, but  $\omega$  is required to occur (strong until).

Note that the between and after-until scopes consider repetition as well, whereas before and after only consider the first instance of proposition  $\rho$ .

With both the class and the scope known, a formal representation of the pattern can be constructed. However, further parameters can be provided for timed and probabilistic aspects. In particular, for ordered classes time bounds can be provided, and a pattern can be made probabilistic by asserting the bound on the probability of the pattern being true.

The patterns permit various logics to be used depending on the underlying model. In the scope of COMPASS, these are LTL, CTL, MTL and CSL, for discrete, timed and probabilistic models respectively.

The actual formalization of patterns is handled via lookup-tables, which are presented in section B.2. Though some general intuition can be given of the formalization, as presented in the description of classes, the combination of class and scope is hard to automate. Note that the amount of useful combinations is limited, allowing this to remain practical in use. An overview of those supported by COMPASS is given in Table 3.1. Note that for probabilistic patterns not all combinations are supported.

Each pattern can be described in natural language, which makes them much easier to understand. In [10], a grammar has been defined which can use phrases expressed in the English language to derive instances of patterns, an approach used in COMPASS as well to save and load properties. This grammar, called the structured English grammar, contains a phrase for each class and scope, as well as the various parameters that they may be associated with. Text used to fill placeholders is delimited to allow the pattern to be unambiguously parsed. The grammar itself is provided in section B.1.

Taking for instance the GPS system described in Listing 2.3, a possible requirement could be formulated as “After the primary GPS receiver has lost its fix, whenever its signal strength exceeds 10, eventually a fix is reacquired.” Here, both the *response*

TABLE 3.1 – Supported classes and scopes in COMPASS.  $\bar{p}$  means probabilistic patterns are not supported for that combination.

Class	Global	Before	After	Between	After-Until
Universality	✓	✓	✓	✓	✓
Absense	✓	✓	✓	✓	✓
Existence	✓	✓	✓	✓ $\bar{p}$	✓ $\bar{p}$
Recurrence	✓	✓	✓	✓	✓
Precedence	✓	✓ $\bar{p}$	✓ $\bar{p}$	✓ $\bar{p}$	✓ $\bar{p}$
Reponse	✓	✓ $\bar{p}$	✓	✓ $\bar{p}$	✓ $\bar{p}$
Reponse-Invariance	✓	✓	✓	✓	✓
Until	✓	✓ $\bar{p}$	✓ $\bar{p}$	✓ $\bar{p}$	✓ $\bar{p}$

class and *after* scope can be applied. The propositions for the response class would then be  $\varphi = \text{primary.signal} > 10$  and  $\psi = \text{has\_fix\_prim}$ . For the scope then  $\rho = \neg \text{has\_fix\_prim}$ . Using the grammar from section B.1, this would be phrased as *After {not has\_fix\_prim}, if {primary.signal > 10} has occurred then in response {has\_fix\_prim} eventually holds..*

#### Property-based Approach

A further approach is based on the use of *properties*— i.e. design-attributes— specified on the system. The idea is to integrate the specification of requirements directly into the model of the system by means of attributes within this model. Formal properties are *extracted automatically* from these properties, permitting the verification of the requirements. The implementation of this approach used by COMPASS is considered separately and in more detail in Section 3.4.

#### Other Linguistic Techniques

A few studies have looked into approaches that distill formal specification from natural language automatically. One branch uses specific subsets of natural language, controlled natural language (CNL), that have a precise mapping. The grammar mentioned in Section 3.3.1 is such an approach. Similar work can be found in [109] and [63].

Other approaches translate requirements based on fully automatic recognition of natural language descriptions, i.e., natural language processing (NLP). In [89], followed by [90], linguistic techniques are used to extract terms from requirements to build an ontology for use with message sequence charts (MSCs), based on named entity recognition (NER). In [61] a tool named NL2ACTL is used for NLP of requirements, representing them in the tree-based logic action-based CTL (CTL) [111]. [8] provides the tool CIRCE for this purpose.

### 3.3.2 MODEL CHECKING

The formal specification of requirements alone permits them to be *validated*, such as checking for inconsistencies. Such a check can be performed by checking whether the conjunction of two requirements is unsatisfiable. However, for the *verification* of requirements, a (design) model is required for the system against which the requirements are verified. In COMPASS, this role is fulfilled by the model specified in SLIM.

Verifying requirements then becomes a model-checking problem, checking that the model satisfies the formal property. In the simplest cases, this is either a yes/no answer. Alternatively, some quantitative measure can be determined, e.g., probabilities. Model checking in COMPASS is discussed in Chapter 5.

These models also let themselves naturally be refined by means of defining subcomponents for systems which prior were still abstract. The following deals with the refinement of requirement based on contract-based analysis (CBA), the implementation of which is treated in Chapter 5.

### 3.3.3 REQUIREMENTS REFINEMENT AND CONTRACT BASED DESIGN

As previously mentioned, during project development requirements are refined into requirements of lower levels of abstraction. This is a common source of errors, such as inconsistent requirements or refined requirements not fulfilling the higher level requirement. Thus, being able to validate refined requirements is certainly beneficial.

Within the component-based methodology of COMPASS, properties can be specified against the interface of components. As systems are further decomposed into subcomponents, associated requirements can be refined into specifications for the subcomponents. Between each level, this refinement can be validated with a *contract-based* approach. Such a contract-based design (CBD) [46, 104] fits well within the hierarchical specification of SLIM models.

Contracts are specified on the interface of a component, and are structured into an *assumption* and a *guarantee*. The assumption asserts the environment fulfills certain conditions. The guarantee specifies how the component must respond, i.e., what its behavior is. Formally, such contracts are denoted as  $\langle \alpha, \beta \rangle$ , where  $\alpha$  is the assumption, and  $\beta$  the guarantee, both being properties specified on the interface of a component.

A contract  $\langle \alpha_2, \beta_2 \rangle$  is said to refine a contract  $\langle \alpha_1, \beta_1 \rangle$  if and only if  $\beta_2$  refines  $\beta_1$  and  $\alpha_1$  refines  $\alpha_2$ . The refinement of properties is expressed in terms of *traces*. Each property is associated with a set of traces, where a trace is a sequence  $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \xrightarrow{e_3} \dots$ , with each  $s_i$  being a configuration of the EDA associated with the component, and  $e_i$  some (internal) event or time delay. For a given property  $\varphi$  this set of traces  $\text{Tr}(\varphi)$  is exactly the set of traces that satisfy the property. A property

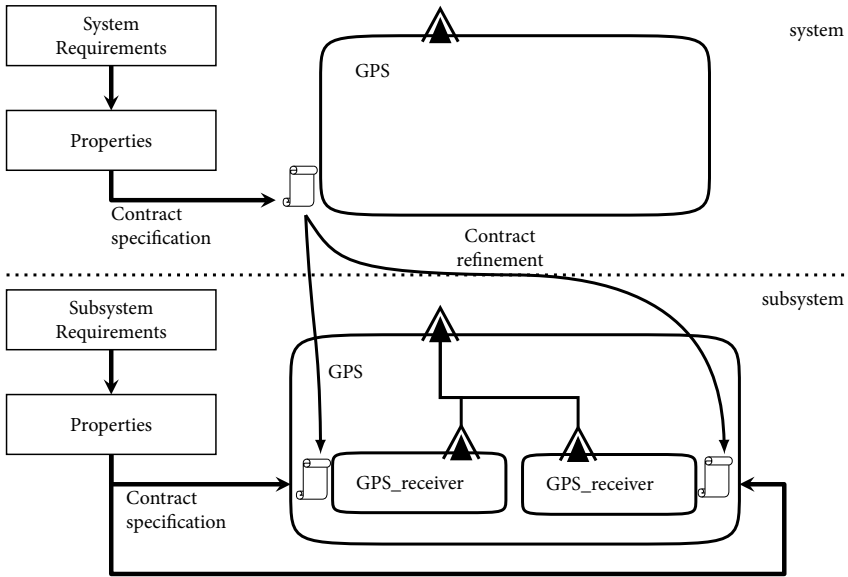


FIGURE 3.3 – Overview of contract based refinement. Formal properties are derived from requirements and used to specify contracts. Systems are refined into subsystems, for which in turn contracts are specified as well. These contracts then refine those of the system.

$\varphi_2$  is said to refine a property  $\varphi_1$  if its set of traces is contained within that of  $\varphi_1$ , i.e.,  $\text{Tr}(\varphi_2) \subseteq \text{Tr}(\varphi_1)$ . For example, a property  $\Box(\Diamond a \wedge \Diamond b)$  is refined by  $\Box\Diamond a$  and  $\Box\Diamond b$ .

Requirements that specify constraints in the inputs of systems can be mapped to assumptions. A common source is interface requirements that specify what the system must, or must not, accept. The requirements that specify how the system should function, in particular what outputs should be provided are mapped to the guarantees. When a requirement is refined, it leads to assumptions and guarantees on subsystems. The refinement can then be validated by checking whether the contracts specified on the subsystems refine that of the system itself. The implementation of this approach in COMPASS is discussed in Chapter 5. A graphical depiction of this process is shown in Figure 3.3. For instance, the contract of the top-level system GPS could state  $\langle \top, \Box\Diamond position \rangle$ <sup>1</sup>. This contract can be refined by the contracts of the GPS\_receiver subsystems, which may state  $\langle \top, \Box\Diamond position \rangle$  as well. Thus, the contracts of the subsystems can be used to prove the GPS system will have a fix.

<sup>1</sup>As no assumption is required, it can be defined as  $\top$ .

### 3.4 CATALOGUE OF SYSTEM AND SOFTWARE PROPERTIES

The taxonomy of Section 3.2 can be used to look at formalizing requirements from a different perspective. Instead of looking for patterns in the way requirements are formulated, it is possible to look at the design attributes of the system that requirements impose their conditions and limitations on. As a system is further refined and expanded, such attributes are specified to configure parts of the system or specify some measures. For example, design attributes may specify scheduling constraints or the boundaries of system parameters.

This provides an opportunity to improve the way models and requirements are formalized. Such design attributes are often specified early on in the design of the system, allowing the formal process to be integrated earlier and with less effort in the development process.

This idea led to the development of the CSSP [HB2], which is a collection of formal properties that are associated with such design attributes. Providing a formal representation of these properties provides a way to formally capture the meaning of the design attributes. One or more properties can automatically be mapped to a formal specification of the associated attributes, which can subsequently be used for model checking, checking the consistency of requirements or CBA.

#### 3.4.1 PROPERTIES AS DESIGN ATTRIBUTES

When specifying requirements for physical system, they commonly refer to certain *properties* of the system, from here on referred to as *design attributes*. Examples of such attributes are response times and mode invariants.

These design attributes can be applied directly to the model representing the system, the same as mentioned in Section 3.3.2. Elements of this design model are assigned values to their specific design attributes (e.g., modes are assigned invariants) based on the relevant requirements (such as state requirements).

As part of the Catalogue of System and Software Properties (CATSY) project, a catalog has been made of the various attributes that can be found in aerospace projects. From this catalog, formal properties were defined which can automatically be derived from such properties.

Concretely, the CSSP is a collection of properties specified for SLIM, aggregated into a property set. Such properties reflect particular design attributes, and can be used together with other CSSP properties to derive one or more formal properties automatically.

Based on the taxonomy presented in Section 3.2, a number of common properties can be identified in aerospace project requirements. These properties have been taken as a basis to define the CSSP property set that will be introduced in Section 3.4.2. Examples of such properties are *monitored parameters* (as can be found in monitoring requirement types), *latency* (performance requirements) or *mean time to failure (MTTF)* (reliability requirements).

Such properties can either be mapped to CSSP properties directly, or be represented by modeling them in the system specification. In section C.4, the design attributes known by the CSSP approach are identified, classified according to requirement types.

#### 3.4.2 CSSP PROPERTY SET

Based on the aforementioned aerospace related properties, the CSSP property set has been defined for SLIM, which is used for the automatic derivation of formal properties. Like other AADL-based properties, they are applied directly to elements of the model.

An example property is `FailureCondition`, which is a property that can be specified directly on components in the model. The type of the property is a list of mode references. When specified, this list contains those modes in which the component is considered to be in a failure state. As will be detailed later, various formal properties can be derived using this information. For example, metrics such as availability can be determined based on this information (by determining the fraction of time spend in non-failure modes versus failure modes). In section C.1, a complete overview is given of the AADL properties defined within the CSSP.

An advantage of this approach is that it is easily extensible. When further design attributes need to be captured by the CSSP, new properties can be defined in its property set. A domain expert can define the associated formal properties, which can subsequently be used by users who are not experts in formal methods. The following section defines the formal properties that can currently be derived from the CSSP.

#### 3.4.3 CSSP DERIVED PROPERTIES

Completing the CSSP approach, formal properties can be derived from the CSSP properties. Each individual CSSP property models a certain *expectation*, which combined with others provides a means to encode them in a formal specification. The mapping between CSSP properties and formal properties is many-to-many, as multiple CSSP properties may be involved in specifying formal properties, some optionally, and multiple formal properties may refer to the same CSSP property.

An overview of the possible formal properties that can be derived from the CSSP is given in section C.2. Such properties apply to a single element from the model, as is indicated by the parameter.

For example, some requirement may specify that an alarm must be triggered upon entering a certain failure condition. This requirement thus specifies the *failure detection delay* attribute for this failure condition. Two CSSP properties allow this to be specified. First, the model must specify some port that is used to represent the alarm which should be raised. Then, the `FailureCondition` property can be specified for this alarm port, which specifies under which condition it must be triggered. Next,

```

device Watchdog
  features
    watchdog_event : in event port;
    anomaly_detected : out event port;
  properties
    CSSP::Timeout => 20 Sec applies to anomaly_detected;
    CSSP::TimeoutReset => reference(watchdog_event)
      applies to anomaly_detected;
end Watchdog;

```

Listing 3.1: Example watchdog device with CSSP properties.

the **AlarmDelay** property specifies the maximum delay, as specified by the *failure detection delay* attribute. Finally, two formal properties can be derived according to section C.2: **CompleteAlarmProperty**(p) and **CorrectAlarmProperty**(p). Verifying both allows one to verify that the specification fulfills the original requirement.

As a more concrete example, consider a watchdog that monitors some signal, and fires an event if that signal does not occur within some predetermined time-bound. In terms of the GPS example from Chapter 2, it could monitor the `position` port, and trigger some event if it is not updated on-time. An example specification is given in Listing 3.1. It specifies two properties, namely **Timeout** and **TimeoutReset**. Both are applied to the `anomaly_detected` port. As indicated in section C.2, these properties suffice to define the *CompleteTimeoutProperty* and *CorrectTimeoutProperty* for this port, which correspond to the formulae  $\Box \Diamond_{\leq 20}(\text{anomaly\_detected} \vee \text{watchdog\_event})$  and  $\Box(O_{\leq 20} \text{watchdog\_event} \rightarrow \neg \text{anomaly\_detected})$  respectively.

### 3.5 DISCUSSION

When it comes to the development of any project, and aerospace projects in particular, the importance of requirements engineering should not be underestimated. Although an important aspect of requirements is answering what is to be done, ensuring the answer is valid is equally as important.

Both validation (doing the right thing) and verification (doing the thing right) of requirements is non-trivial. Doing so by means of formal methods provides hard guarantees that this is done correctly, avoiding some costly pitfalls. However, this is not simple to do, and various aspects and approaches have to be taken into account.

Various logics provide different degrees of expressiveness and support for different classes of models. However, support for more complex expressions or semantics often comes at the cost of a higher complexity for solving satisfiability or model checking problems. Therefore, it makes sense to adapt the choice of logic to the specific query imposed by a requirement, rather than going for a single all-encompassing solution. However, this imposes a higher burden on the framework chosen for formalization.

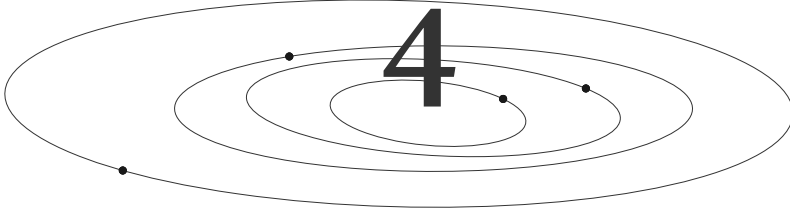
Formalizing requirements by hand is known to be difficult, and various degrees of automation exist to alleviate this. Pattern-based approaches are common and cover many types of requirements. Mapping them to natural language is possible, though restricted to a limited set of grammatical structures. Full natural language processing can improve this, but comes at a significantly higher complexity.

The CSSP provides an approach that encodes requirements directly into the model of the system being developed. Although this requires such a model to exist prior to translating the requirements, it provides a simple structured way of formalization. The current set of supported properties is very specific to the aerospace domain it was originally designed for, but can easily be extended to others. Unfortunately, no case study has yet been performed that makes extensive use of the approach, making it difficult to estimate its effect on the validation of requirement. This remains to be future work.

The possibilities for specification and analysis of requirements addressed in this chapter can be summarized as follows: First, the taxonomy presented in Section 3.2 can be used to identify the type of a particular requirement, which influence the parameters that drive the analysis.

Next, the requirement can be formalized into a formal property. Pattern-based approaches allow for selecting a pre-formatted structure with placeholders, that can be filled in according to the requirement. An attribute-based approach instead inserts properties into the model specification. Both approaches automatically derive a formal property from the specification. Otherwise, a formal logic may be used (such as one from Appendix A), but this requires expert-level knowledge of these logics that the other approaches avoid. A possible workflow will be discussed in Section 5.3.1.

Finally, properties can be used to specify contracts for the various systems and subsystems in the model, permitting the use of CBA. This allows for a compositional-based approach, which helps tracking the refinement of requirements across project abstraction levels, as well as enabling compositional-based reasoning, the latter making formal analysis tractable for a larger class of model than monolithic approaches may be able to handle.



## STATISTICAL MODEL CHECKING

*ABSTRACT – This chapter introduces the principles of SMC, some of its benefits and shortcomings, as well as its application within the COMPASS toolset by means of the SLIMSIM tool. Two key advantages make SMC very useful in practice: it can handle a very large class of input models, and it scales well with regard to the complexity of the input model.*

---

**M**OST commonly when people refer to model checking, it is implicitly assumed to be of exact or numerical nature. Most model checkers use such methods to provide hard guarantees about their results. An unfortunate consequence is that such methods are often computationally very hard, or even undecidable. An alternative approach uses statistical methods, which make use of simulation in order to decide whether some property holds or not. Such outcomes come with a certain statistical significance, determined by the thoroughness of the simulation. This means that a statistical model checker will not be able to provide the absolute results that conventional (i.e., analytical or numerical) model checkers would. The advantages of statistical methods are, however, significant. As the only requirement is that the model can be simulated—in other words, have measurable execution paths—a much larger class of models can be handled by statistical model checkers. The approach is also highly parallelizable, taking the benefit from modern multi-core architectures. Finally, the demand for system resources is generally more favorable. A statistical model checker only needs to keep track of the current simulation state, and does not need to build the complete state space, or part thereof.

As the SLIM language permits the specifications of models that contain discrete, continuous and probabilistic behavior, the need for statistical model checking for *quantitative analysis* becomes apparent, as neither numerical nor analytical methods are available that can handle this class of models. Prior to the use of statistical model checking, probabilistic analysis was limited to those models that did not spec-

ify any continuous dynamics, or even infinitely sized models. To fulfill this need, a statistical model checker named SLIMSIM was created, which supports the analysis of quantitative properties for SLIM models without any restrictions regarding their specification.

#### 4.1 ACCURACY AND PRECISION

Two key concepts play a role when discussing statistical methods for model checking: accuracy and precision. Accuracy pertains to how close an answer is to the truth. In qualitative model checking, answers are accurate if and only if the model-checking algorithm results in true if and only if the property is true. For quantitative model checking, it is the distance between the calculated probability of the property being true and the actual probability of the property holding true.

Precision pertains to the closeness of different results under the same conditions. More specifically, it describes either repeatability or reproducibility. In the scope of quantitative model checking, usually repeatability is meant. A precision of  $10^{-6}$  would mean the results has an absolute error of at most  $10^{-6}$ .

All prior mentioned model checking methods have either been analytical or numerical in nature. Analytical methods can provide answers that are guaranteed to be correct, and with infinite precision. For instance, analysis on probabilistic programs may generate expressions over the expected runtime of a program [85]. Numerical methods also provide guarantees, but at a (arbitrarily) limited precision. Many model checkers for Markov-models support some variant of value iteration, which can determine metrics such as a reachability probability or expected time up to a certain precision (intuitively, values are updated until their change is within a certain threshold). Better precision generally makes the problem computationally harder. Thus, a tradeoff is required between the preciseness of the result, and the time and memory available to calculate it.

statistical model checking relaxes this by using statistical means to derive an answer, meaning not only precision, but also accuracy is limited. Statistical approaches have an inherent amount of error in their results, which is expressed in the *confidence*. This is the probability of the result obtained by simulation being true. For qualitative analysis, this involves the probability of either a false positive or a false negative. For quantitative analysis, it is the probability that the calculated result coincides with the actual probability. For quantitative algorithms, the *error bound* additionally specifies the size of the range in which the result is deemed true. Both metrics can be chosen arbitrarily within the range  $[0, 1)$ , with similar trade-offs as for the precision of numerical methods.

## 4.2 STATISTICAL ANALYSIS

SMC [67, 142] techniques are based on the Monte-Carlo method, and extend it to be able to verify *bounded* temporal properties<sup>1</sup>. Effectively, the simulation “executes” the model, and checks at runtime if the property holds.

The Monte-Carlo method [110, 135] was originally invented around 1946, providing a way to tackle problems that could not be solved by deterministic methods. It can be used for problems of a probabilistic nature. For such a problem, a random set of inputs is sampled, to which then a deterministic computation is applied (i.e., to check whether they lie in a certain set). These results can then be used to provide a solution for this problem.

In the case of SMC, the problem can be stated as determining if some property holds true with some probability above or below some threshold. To estimate this using the Monte-Carlo method, samples are generated in the form of traces from the transition system described by the model. The outcome of each generated trace is used to determine the likelihood that this is the case.

A variant thereof determines for probabilistic properties the actual probability. Samples are generated until some threshold is exceeded, at which point the mean of the samples can be used to estimate the probability.

Taking the above into account, any SMC algorithm relies on three components:

- » A path generator (discrete event simulator);
- » A property checker;
- » Some termination condition.

The property checker decides whether the path generation needs to continue or not. It checks at runtime if the property holds, does not hold or if more simulation steps are required in order to draw a conclusion. This also means it can only verify properties that only require a bound (but arbitrary) number of steps to verify. Otherwise an infinite number of steps may be required.

The termination condition is necessary to determine the number of traces that need to be generated. Normally, some metric is used which either decides a priori the number of traces to reach a statistically significant conclusion, or decides on the fly if enough traces have been collected, which may vary depending on the variation of the outcome.

The two SMC algorithms can be classified as *probability estimation* and *hypothesis testing*. The latter formulates a hypothesis that the estimated probability is above or below some threshold probability  $\theta$ . This is relevant for many probabilistic model checking properties, which encompass queries such as “does the probability of occurrence of  $x$  exceed  $\theta$ ?”

---

<sup>1</sup>Properties that can be checked in a finite number of steps or with a finite amount of time.

TABLE 4.1 – Overview of hypothesis test algorithms.

Name	Sample size	Parameters (excluding $\alpha, \beta$ )
Gauss-CI	Fixed	$N$
Chernoff-CI	Fixed	$\epsilon$
Chow-Robbins	Mixed	$\epsilon$
SPRT	Sequential	$\delta$
Gauss-SSP	Fixed	$\delta$

Probability estimation on the other hand simply determines the probability of the property holding true. In other words, it tries to answer questions like “what is the probability of occurrence of  $x$ ?”

#### 4.2.1 HYPOTHESIS TESTING

In the case of hypothesis testing, the problem is formulated using a *null hypothesis*  $H_0$  and one or more *alternative hypotheses*  $H_n$ . The idea is to collect enough samples to allow either  $H_0$  to be rejected (accepting one of  $H_n$ ), or vice versa. Note that neither type of hypothesis can be *proven* to hold, but only be given a certain confidence. The latter is achieved by limiting the probability of drawing a conclusion erroneously. One possibility is to reject  $H_0$  incorrectly, which is an error of *type I* — a false negative. The probability of a type I error  $\alpha$  is the *significance* level. The other is to accept  $H_0$  when in truth some  $H_n$  holds true, which is an error of *type II* — a false positive. Having  $\beta$  be the probability of a type II error,  $1 - \beta$  is the *power* of the hypothesis test.

In the setting of model checking, the question is generally whether or not the true probability lies above or below some threshold. This can be formulated by means of a null hypothesis  $H_0 : p = \theta^2$  and alternative hypotheses  $H_+ : p > \theta$  and  $H_- : p < \theta$ . In order to determine which of the hypotheses to accept, the number of positive samples either has to be above some threshold to accept  $H_+$ , or below another to accept  $H_-$ . Alternatively, it may happen that such a threshold is never reached, at which point the test becomes inconclusive.

A variety of algorithms exist that can perform hypothesis testing [120]. They vary in terms of achievable power, significance or the number of samples required for drawing a conclusion of a particular power and significance.

A summary of those tests can be found in Table 4.1. They hint towards their sample sizes and what input parameters are expected. For the prior, *fixed* means the sample size is defined a priori, *sequential* means the decision is made after each sample, and *mixed* means a decision is also made after each sample, but with some known upper-bound. What follows is a summary of the tests from [120] which are most relevant for the discussion that will follow.

<sup>2</sup>Note that  $p = \theta$  can never be shown to be correct, but the inverse is possible.

In the following,  $N$  refers to the number of samples. Each sample is drawn from a Bernoulli process with  $X_i, i \in [1..N]$  random variables, each variable representing the outcome of testing a property  $\varphi$ , where  $X_i = 1$  means  $\varphi = \top$  and  $X_i = 0$  means  $\varphi = \perp$ . Finally,  $S_N = \sum_{i=1}^N X_i$  gives the estimator sum.

**Gauss-CI** The Gauss-CI test is based on the central limit theorem (CLT). The sample size  $N$  is fixed beforehand, from which a confidence interval can be derived. The variable  $N$  is determined such that  $\Pr[S_N > u^* \mid H_0] < \alpha$ , where  $u^*$  defines the boundary between accepting  $H_+$  and not being able to draw a conclusion yet (i.e., not rejecting  $H_0$ ). In other words, if  $\phi$  is the standard normal cumulative distribution, then

$$u^* = \phi^{-1}(1 - \alpha)\sqrt{N\theta(1 - \theta)} = -l^* .$$

Here,  $l^*$  is the lower bound where  $H_-$  can be accepted.

**Chernoff-CI** The Chernoff-CI test uses a different confidence interval, and uses the Chernoff-Hoeffding (CH)-bound to determine the number of required samples. Choosing  $\alpha$  and an *approximation parameter*  $\epsilon$ , the number of samples to draw is then determined as

$$N = \frac{1}{2\epsilon^2} \log \frac{2}{\alpha} .$$

It then holds that

$$\Pr[|\bar{X} - \theta| \geq \epsilon] \leq \alpha .$$

After sampling  $N$  times, if  $|\bar{X} - \theta| > \epsilon$ , where  $\bar{X} = \frac{1}{N}S_N$ , then  $H_+$  or  $H_-$  is accepted depending on whether  $\bar{X} > \theta$  or  $\bar{X} < \theta$ . If that is not the case, the test is inconclusive.

**Chow-Robbins** The Chow-Robbins test is based on the Gauss-CI test. However, as samples are generated the width of the confidence interval for  $\hat{p}_N = \frac{S_N}{N}$  is determined. At  $N'$  samples this width is  $2\phi^{-1}\alpha\sqrt{\hat{p}'_N \frac{(1-\hat{p}'_N)}{N'}}$ . If the confidence interval is then entirely above  $\theta$ ,  $H_+$  can be accepted with confidence  $1 - 2\alpha$  (and vice versa for  $H_-$  when below  $\theta$ ). This allows the test to terminate possibly before generating  $N$  samples.

**SPRT** The sequential probability ratio test (SPRT) test [142] is based on the work of Wald [140]. The idea is to calculate the ratio

$$\frac{p_+^{S_N}(1 - p_+)^{N - S_N}}{p_-^{S_N}(1 - p_-)^{N - S_N}} ,$$

where  $p_+ = \theta + \delta$  and  $p_- = \theta - \delta$ , with  $\delta$  being a parameter specifying the indifference level. If the above ratio is large,  $H_+$  is accepted, and if it is small  $H_-$ . To determine

the boundaries at which the ratio becomes large, respectively small enough, the following can be used:

$$u(N) = \frac{1}{q_1} (\log u' - q_2 N) - N\theta$$

$$l(N) = \frac{1}{q_1} (\log l' - q_2 N) - N\theta,$$

where

$$q_1 = \log \left( \frac{p_+ \cdot (1 - p_-)}{(1 - p_+) \cdot p_-} \right), q_2 = \log \left( \frac{1 - p_+}{1 - p_-} \right)$$

and

$$l' = \frac{\alpha_1}{1 - \alpha_2}, u' = \frac{1 - \alpha_1}{\alpha_2}.$$

**Gauss-SSP** The Gauss-SSP test can be compared to the SPRT test, but with a fixed sample size  $N$ . After sampling  $N$  times, if  $S_N > N\theta$  then  $H_+$  is accepted, otherwise  $H_-$ .  $N$  has to be chosen large enough according to  $\alpha$ , for which the following equations apply:

$$N \geq \left( \frac{\phi^{-1}(1 - \alpha_1)}{\delta} \right)^2 (\theta - \delta)(1 - \theta + \delta),$$

$$N \geq \left( \frac{\phi^{-1}(\alpha_2)}{\delta} \right)^2 (\theta + \delta)(1 - \theta - \delta).$$

Here,  $\alpha_1$  is the probability of a false positive when accepting  $H_+$ , whereas  $\alpha_2$  is the probability of a false positive when accepting  $H_-$ .

#### 4.2.2 PROBABILITY ESTIMATION

Probability estimation concerns itself with the *quantitative* aspect of probabilistic model checking, where the probability of a property holding true ( $\Pr[\phi] = p$ ) is estimated. Such an estimation can be acquired by generating  $N$  samples, and taking the ratio  $s_N/N = p'$ . The question remains on how to determine  $N$ . To this end, the confidence interval (CI) methods can be used, as the CI gives a direct quantitative measure. This makes the Gauss-CI, Chernoff-CI and Chow-Robbins approaches suitable. For example, the CH-bound can be used as follows. Given the parameters  $\epsilon$  and  $\alpha$ , recall the number of samples can be determined by  $4 \log(\frac{2}{\alpha}) / \epsilon^2$ . Then, the probability is estimated such that  $\Pr[|p' - p| \leq \epsilon] = 1 - \alpha$ .

#### 4.2.3 RELATED WORK AND TOOLS

A variety of tools exists, each tool supporting various modeling formalisms and statistical model checking approaches. Most tools focus on hypothesis testing, although a few support probability estimation as well.

The choice of statistical method used depends on the domain of the problem and which parameters are most relevant. Fixed sample sizes allow the computational effort to be determined up front, but lack the ability to terminate earlier when the result can be decided beforehand. On the other hand, sequential tests may need an arbitrarily large number of samples. However, in all cases it is important to consider what the driving parameters are. Some amount of error has to be accepted, but the type depends on the approach used. Finally, for problems where the estimated probability lies close to the true probability, CI methods generally do not work very well, the result being either inconclusive or requiring a impractical number of samples. An overview of these effects can be found in [120].

YMER [143] supports qualitative analysis using SPRT for discrete-time Markov chains (DTMCs) and continuous-time Markov chains (CTMCs), as well as generalized semi-Markov processes (GSMPs). MRMC [87] supports DTMCs, CTMCs and Markov reward models (MRMs), and can perform qualitative analysis using CI-based approaches. Gauss-single sample plan (SSP) and the Chow-Robbins (CR)-method. An extension PVEStA adds support for parallelization. It further evolved with MultiVeStA [124], adding support for multiple simulation engines.

On the quantitative side, a few tools exist as well. APMC [78] support statistical analysis for DTMCs, using an approach based on CH-bound. COSMOS [14] supports stochastic Petri nets (SPNs) based on the CR test for probability estimation.

Both quantitative and qualitative analysis is supported by UPPAAL(-SMC) [53] (version 4.1.14 and up), PLASMA-LAB [31], PRISM [95] (version 4.0) and MODES [27]. UPPAAL(-SMC) can verify properties for priced timed automata (PTA), using the SPRT approach for qualitative purposes, the CH-bound for quantitative purposes. PRISM supports DTMCs, CTMCs and Markov decision processes (MDPs). For quantitative analysis, confidence interval and CH-based approaches are available, with additionally SPRT for qualitative analysis. PLASMA supports DTMCs, MDPs and, with an extension, PTA. MODES is part of the MODEST toolset, and supports the analysis of stochastic timed automata (STA) using either SPRT, CI or CH methods.

### 4.3 CAVEATS IN SIMULATION SEMANTICS

For fully probabilistic models, statistical model checking is a very useful approach. However, employed in the context of the models treated in this thesis, some caveats need to be taken into consideration [HB1]. These mostly pertain to the problem of *scheduling*. As a statistical model checker requires a fully probabilistic model as input, any non-determinism has to be resolved. The same holds for scheduling concurrent systems. These considerations, as well as some other technical ones, are presented in the following.

#### 4.3.1 TRANSITION INTERLEAVING

For timed models, a particular aspect that needs to be considered is the ordering and interleaving between discrete and timed transitions. Such models do not specify this, and it is up to the analysis engine to explore the different combinations that lead to all possible states. This means that subsequent timed or discrete transitions are possible. However, for performance reasons a simulator may choose to only consider a single timed transition between discrete transitions. Such an implementation must still consider the possible points in time that may be relevant to the property under investigation, when its truth value may change transiently under the possible range of time delays.

Most often, the timed and discrete steps are considered as integral. This leads to the notion of the one-step probability, which is the probability associated with both waiting a specific delay (which may be none) and choosing a specific discrete transition. With such an approach, the possible combinations must be considered carefully, which is discussed below.

#### 4.3.2 UNDERSPECIFICATION

A major difference between SMC and other model checking techniques is that SMC relies on simulations, which in turn require the model to be — to some degree — concrete. This means that from any given state that the model is in, it must be possible to determine a unique next state.

Generally, as models provide some level of abstraction from a real system, some branching is possible, i.e., from one state multiple next states are reachable. Such branches generally stem from the fact that either it is not known a priori what state the system will be in next, given the current state, or because it is left open to permit different implementations. This is referred to as underspecification of the model, and within the paradigm of COMPASS can take two forms: *non-determinism* and *time bounds*.

If underspecification occurs in a model to which SMC is to be applied, it has to be resolved somehow. Ideally, a single choice would be made. However, this limits the search space. More often, some probability distribution is applied over the possible choices.

The consequence of assuming a probability distribution is that the resulting probabilities do not range from all the probabilities over the choices, but instead are the weighted sum of them, depending on the distribution used. This means that any probability determined by the simulator lies somewhere in the range of the overall minimum or maximum, without knowing how far from either point this probability is.

Ultimately, the engineer has to decide if this approach is acceptable. For systems where the non-determinism has only “minor” influence, it may very well be, but the difference in semantics should be well understood before doing so.

### *Non-Determinism*

Non-determinism applies to the discrete aspects of the model's behavior, where from one state multiple other states can be reached, where it is free to choose which one. In other words, if a state has multiple enabled transitions with the same action, it is said to be non-deterministic. A particular important aspect of this is that it is a purely abstract notion, and hence cannot be simulated. Because the model does not provide any concrete information on what state is next, execution cannot continue.

As an example, see Figure 4.1. Starting in location  $p_0$ , three choices are available. Two transitions synchronize on action  $\alpha$ , and one on action  $\beta$ . The choice between the  $\alpha$  transitions represent an *internal* choice, and the choice between  $\alpha$  and  $\beta$  an *external* choice. External choice influences, or may be influenced by, synchronizing components, internal choices are not.

Multiple ways of handling this issue are possible. First, the simulator may simply reject the model, which for example is the default behavior of the MODEST tool [27]. This ensures the output of the simulator remains sound, as it will not impose any behavior that the model does not specify. It is, however, quite restrictive and prevents the simulator from producing results that may otherwise still be useful.

Another possibility is to try and transform the model such that non-determinism no longer occurs. This is possible by means of slicing away parts of the model that are not relevant to the property being analyzed [112], or reducing the model by means of bisimulation minimization. The latter may remove non-determinism if the possible choices lead to an equivalent outcome. In this case it may be dubbed *spurious* [27].

Finally, the simulator may simply assume a probability distribution over the possible choices. This is a discrete probability distribution ranging over the various possible choices. In COMPASS, the number of choices is always finite, but depending on the model in general this may be either a finite or infinite distribution. Usually, for finite models the discrete uniform distribution is chosen, based on the assumption that without further information, all possibilities can be considered equipossible and hence should all occur equally likely.

### *Time Bounds*

In the case of timed models (or hybrid in general), timed transitions specify the delay between state changes. In the model, this delay may be underspecified, meaning it is left open what value the delay assumes in the model when undertaking the transition. Often, the delay is bounded from above, meaning it has a maximum. A minimum delay may be specified as well. Finally, it may be unbounded, or bounded from below only, meaning a maximum delay has not been specified.

To deal with this, the simulator has to assign a probability distribution, similar as is required for non-determinism. The main difference is that it now is a continuous distribution function, as delays may vary over a continuous range. For bounded

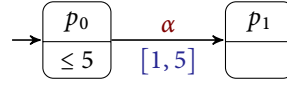
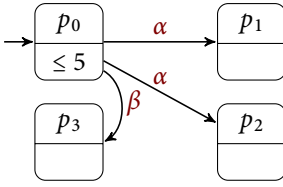


FIGURE 4.1 – Example of non-determinism.

FIGURE 4.2 – Example with time bounds.

delays, generally the (continuous) uniform distribution is assumed for the same reasons as for non-determinism. For unbounded delays, a distribution with infinite support is required. It depends on the semantics of the model what may fit best, though in the aerospace domain this is often the exponential distribution. This distribution is frequently used to model, e.g., failure rates, and fits them well in practice. As there is often no upper bound on the time until failure, an upper bound cannot be specified. However, failure rates are often already described by means of an exponential rate, and this information may be embedded in the model directly.

An example is shown in Figure 4.2, where a transition is enabled between one and five time units. Furthermore, the location  $p_0$  has an invariant that does not permit time to exceed five while it is active. Thus, some distribution is required that supports values ranging from zero to five, or a subset thereof.

#### 4.3.3 SCHEDULING

In the case of a model that contains more than one process, the scheduling between these processes is often left unspecified. Such an underspecification makes it possible for analysis to examine all possible scheduling options, but leaves the same problem as mentioned before in this section.

For timed models, the scheduling may be based on the notion of the ‘fastest’ process, which is the process for which the shortest delay has been sampled. If the delay and state change are considered as a single step — a concept which is further discussed in Section 4.3.1 — this solution comes forward naturally.

Alternatively, and required for discrete models, the process to execute has to be selected probabilistically. Again, based on equiprobability, the uniform distribution is a good candidate.

#### 4.3.4 SYNCHRONIZATION

When two synchronized processes have the same action transition enabled, care has to be taken that both processes remain in a consistent state after the transition has been taken. Foremost, it has to be ensured that the (possible) invariants for each process are not violated. However, it also affects the possible samples for the delays that are associated with the transitions. If the ranges of the possible delays

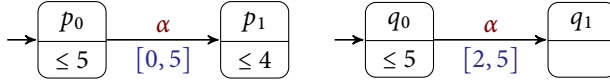


FIGURE 4.3 – Example network with synchronizing actions.

are not identical and one-step semantics are employed, the intersection of these ranges should be considered, otherwise at least one transition would not be enabled. An example is shown in Figure 4.3. Both  $p_0$  and  $q_0$  do not permit time to exceed five. Synchronizing on action  $\alpha$ , both transitions can only be taken in the interval  $[0, 5] \cap [2, 5] = [2, 5]$ . Furthermore, the target location  $p_1$  does not permit time to exceed four, hence the transition can only be taken in the interval  $[2, 5] \cap [0, 4] = [2, 4]$ .

#### 4.3.5 BIAS

Both inter- and intra-process delay sampling may be subject to a bias depending on the semantics being used. First of all, the order between selecting a delay and a non-deterministic transition greatly matters. When non-determinism is resolved first, the choice of transition is based solely on the distribution between transitions, normally uniform. On the other hand, if the delay is sampled first, the size of the time ranges of each transitions adds a bias towards transitions with larger ranges (see also Section 4.4).

Bias may also be introduced if the generation of a delay is possible at which time a transition is not enabled. For non-convex intervals, a sample that falls outside of the interval, but is larger than the interval's minimum, generates a bias towards higher delay values.

Consider the examples in Figure 4.4. Determining first which transition to take, in all three cases the distribution between the two target locations is the same (i.e., 50/50). However, determining the time delay first some differences become apparent. Assuming a uniform distribution in the interval  $[0, 5]$ , the first automaton  $P$  has a 0.8 probability of moving to  $p_1$  and 0.2 for  $p_2$ . The inverse holds for the second automaton  $Q$ , as any delay in the interval  $(1, 4)$  will eventually lead to a total delay within  $[4, 5]$ . The rightmost automaton  $R$  shows that the delay itself is not enough, as any delay in  $[4, 5]$  leaves the choice for either transition open. Selecting uniformly, the distribution would be 0.9 for  $r_1$ , and 0.1 for  $r_2$ . It is exactly this behavior that leads to the discussion of strategies presented in Section 4.5.

#### 4.3.6 STRICT BOUNDS

Invariants and transition guards may specify expressions over clocks by means of strict comparison operators. This results in open ranges for the intervals that govern the possible delays. For instance, an invariant may specify a bound  $< 5$ , which permits delays up to but not including five units of time. For practical implementa-

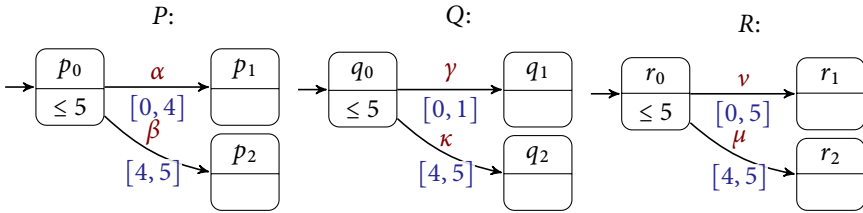


FIGURE 4.4 – Examples of underspecification of both choice and time with convex, non-convex and overlapping intervals respectively.

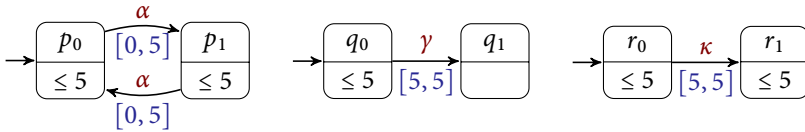


FIGURE 4.5 – Examples showing respectively a time-lock, an action-lock and a deadlock.

tions, such ranges cannot be considered as-is, as a delay has to be a concrete value, and is stored with a fixed width. Either the simulator will have to reject such models, or approximate the range by adding or subtracting the smallest representable unit from the range.

#### 4.3.7 INVALID PATHS

In some cases it is possible for the model to permit certain behavior with unsound or unwanted semantics. In COMPASS, states that exhibit such behavior are either *time-locks*, *action-locks* or *deadlocks*. A time-lock (or *Zeno-state*) is a state from which no path emerges that allows time to progress indefinitely. That is, any infinite execution from this state allows only a finite amount of time to pass. Such executions are called *Zeno* [134]. The states of the first automaton in Figure 4.5 are all time-locks, as none allow time to progress beyond five units.

Action-locks are states in which no further discrete behavior can occur, but time is permitted to progress. It depends on the semantics of the model if this can be considered permissible. However, within the aerospace domain, normally reactive systems are modeled, for which this is invalid behavior. The second automaton of Figure 4.5 shows such states in location  $q_1$ : No other locations can be reached, though time is permitted to increase.

Finally, deadlocks are a combination of time- and action-locks, where no behavior is possible at all. The third automaton in Figure 4.5 has a deadlock in location  $r_1$ , where after a total delay of 5, neither time can increase, nor can any location be reached by a discrete transition.

When such states can be reached it is considered a modeling error, but such errors cannot always be detected up front. For the simulator, action- and deadlocks simply

cause the simulation to terminate. Zeno executions pose a larger problem: When not detected, it may cause a simulation to never terminate assuming the time bound is not reached for the property under investigation.

Checking for states that exhibit Zeno behavior is a hard problem. Non-Zenoness can be guaranteed by certain properties [77], but this alone is not sufficient, as for a simulator a state needs to be *Zeno-free*. Alternatively, syntactic properties can be specified that ensure that locations are *strongly non-Zeno* [30]. However, these may be too strong. In practice, a simulator is often safeguarded against Zeno behavior by having a maximum number of steps configured, which terminates the simulation when exceeded.

#### 4.3.8 PARALLELIZATION

As the core algorithm for SMC requires a large number of statistically independent samples, it allows itself to be trivially parallelized: each processing unit can simply generate samples individually. However, some care has to be taken not to introduce a bias, as demonstrated in [143]. When considering the result of a sample as it becomes available, a bias is generated by samples associated with short paths, as these are generated faster. Some balancing is required in order to avoid such behavior, where each processor is tasked with generating the same amount of samples, thus avoiding the issue. Depending on the SMC algorithm, this number may be known a priori, in which case the number of samples per processor can be determined up front as well. If this is not the case, processors may simply buffer samples up to a fixed amount, until processors all have reached this amount, at which point these samples are considered, similar to a barrier.

#### 4.3.9 RARE EVENTS

A particular challenging aspect of models for SMC are rare events, which represent behavior that has a very low probability of occurring. Measuring for such events using the regular Monte-Carlo method requires a prohibitively large number of samples in order to reduce the variance enough to be able to give meaningful statistical answers.

Various techniques do exist to work around this problem, by introducing a controlled bias towards such rare events, making them more likely to occur during simulation, referred to as *rare event simulation* [121]. Two approaches are well-known, *importance sampling* and *importance splitting*.

Importance sampling[75] adjusts the probability distribution associated with some random variable(s) in the model. The result is a biased estimation, which needs to be corrected by means of weighing it with a *likelihood ratio*. Being able to compute such likelihood ratios is therefore required for importance sampling, though a large variety is known for various stochastic processes.

Importance splitting work by segmenting the reachable states into various levels of importance, each higher level being “closer” to some rare event that is measured.

The probability of the rare event can then be estimated using the conditional probability of reaching each level, given the probability of reaching the prior level. A well-known approach to this is the RESTART method [137]. Effectively, each time a sample reaches a higher level, further samples are generated starting from that point, until it either reaches a lower level again, or a higher level.

The segmentation of states occurs according to some *importance function*. The quality of the analysis depends greatly on the quality of this function. They can either be defined ad hoc by a domain expert, or be derived automatically, e.g., see [37].

## 4.4 CLASSIFICATION OF IMPLEMENTATIONS

Based on the considerations discussed in Section 4.3, some classification can be made based on the possible design choices. As these choices influence the results of the simulator, it is important to keep them in mind when designing a simulator.

### 4.4.1 ORDER

As mentioned in Section 4.3.5, a simulator may choose between first resolving the choice between transitions, or the time delay. The prior is referred to as *Early ordering*, the latter as *Delayed*.

With early ordering, the simulator chooses a transition from each enabled process to execute next, thus choosing the transition purely based on the distribution between them, avoiding any bias that may be introduced by the range of possible time delays for each of them.

Delayed ordering first samples the time delay, and picks a transition that is enabled at that point in time, based on the distribution between those enabled transitions.

A combination is possible as well. In [98], the preselection policy enabled the user to select some transition up front. After sampling a delay, a choice is made between these transition.

### 4.4.2 ACCURACY

In Section 4.3.2, the need for defining a distribution over time delays was noted. This can be done in various ways. One aspect that needs to be considered is whether delays may be sampled at which no transition is enabled, or if the range of possible delays must match exactly those at which at least one transition is enabled.

With *exact* accuracy, the latter is achieved. A delay can be sampled if and only if at least one transition is enabled. This means the simulator must consider the invariants of source and target locations, as well as all the intervals during which a transition is enabled, any of which may be non-convex.

With *approximate* accuracy, such constraints are relaxed, permitting the simulator to consider a wider range of delays. This means a delay may be sampled at which

time no transition can be taken. What kind of over-approximation is considered, as determined by the *strategy*, is discussed in Section 4.5. How to deal with situation where no transition is enabled is discussed in Section 4.4.4

#### 4.4.3 SCOPE

Up to this point, implicitly only a single delay was considered per process. However, a different approach is possible, where a delay is sampled for each possible transition individually, an option available when the ordering is *delayed*. If a single delay is sampled for the current location only, the simulator is said to have a *location local* scope. When a sample is generated for each transition individually, the scope is said to be *transition local*.

With location local scope, the single sample is generated from a distribution that has to account for all the currently possible transitions. After sampling a delay, the transition to be executed has to be selected from those that are enabled after the delay. Although simpler in use, this approach introduces a bias towards transitions with larger time intervals in which they are enabled, and may make selecting a transition with a point interval impossible (probability is zero, though for practical implementations very low).

With transition local scope, after sampling a delay for each individual transition, the transition is selected, possibly by selecting the shortest delay, or by a probability distribution over the transitions.

#### 4.4.4 RACE POLICY

It is possible for a generated sample not to be used for the next state transition due to the existence of a race condition. When this happens, a choice needs to be made over the lifetime of such a sample. Two types of policies control this.

*Memory* policies dictate what happens to the sample if the process lost the race. In [98] three such policies are described:

- » Age memory: This policy retains the sample;
- » Enabling memory: The policy retains the sample as long as the transition remains enabled, and discards it otherwise;
- » Resampling: This policy always discards the sample.

When sampling is performed with approximate accuracy, the *attitude* policies come into play:

- » The *conservative* attitude drops a sample when no transition is enabled after the sampled delay;
- » The *progressive* attitude applies the sampled delay regardless of no transition being enabled.

TABLE 4.2 – Summary of algorithmic policies.

Policy	Transition	Keep sample	Reject sample
Memory	Enabled	Age & Enabling Memory	Resampling
	Disabled	Age Memory	Enabling Memory
Age	Enabled	Progressive & Conservative	-
	Disabled	Progressive	Conservative

If the progressive attitude is followed, a bias is introduced towards higher delays. If a transition is enabled during the interval  $[1, 2] \cup [4, 5]$ , a sample in the range  $(2, 4)$  will automatically imply a total delay in the range  $[4, 5]$ .

These policies are relevant in case a non-memoryless distribution is used. For memoryless distributions such as the exponential distribution function, resampling does not alter the probability distribution.

#### 4.5 STRATEGIES

Based on the accuracy of the approach for simulation, there are various options to determine the interval in which a delay is sampled. Here, four such *strategies* are considered.

- » The *ASAP* strategy selects the earliest possible delay from the possibly enabled transitions. This makes the delay time deterministic, and forces the system to ‘progress’ as fast as possible.
- » The *Progressive* strategy determines the exact interval(s) in which transition are enabled. This, it ensures that after a delay, a transition is always enabled. This corresponds with an exact accuracy.
- » The *Local* strategy uses the invariant of the local state to determine the range in which a sample is generated. This considers all delays that are considered valid by the model, but may easily lead to action- or deadlocks.
- » The *MaxTime* strategy delays as much as possible as permitted by the local invariant. It is the intuitive opposite of the *ASAP* strategy. It is most useful for finding states in which action-locks occur.

The effect each strategy has on the outcome is hard to predict. Although the results are guaranteed to remain within the theoretical minimum and maximum, the difference with each boundary cannot be given. Some work has been done to improve this, such as making use of *reinforcement learning* to approach these boundaries [76], however, no guarantees can be given.

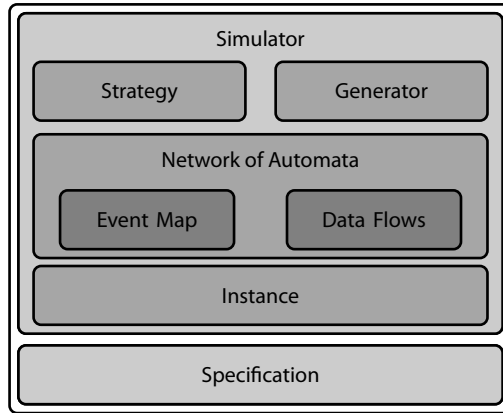


FIGURE 4.6 – Architecture of the simulator.

## 4.6 SLIMSIM

To introduce SMC in the COMPASS toolset, a tool dubbed `SLIMSIM` was developed [HB4]. The architecture of this tool is described in the following. In brief, it reads a preprocessed SLIM file and some properties, and will run a statistical analysis for each given property. The tool itself is written in the C++ language (consisting of about 14K lines of code). An overview of its architecture is given in Figure 4.6.

### 4.6.1 INPUT

The initial step is for COMPASS to load the model, and replace any ‘syntactic sugar’ with basic SLIM constructs. This includes

- » Replacing time units with untimed values;
- » Converting time bounds to clock invariants and guards;
- » Replacing references to constants with their actual values.

Furthermore, to simplify the analysis of time bounds, clauses are generated for invariants and guards that involve timed variables. Such clauses consist of a constant Boolean expression that does not change over time in the same discrete state, and a number of linear equations in the form of  $c = a \cdot x + \dots + b \cdot y$ , where  $x, \dots, y$  are timed variables, and  $a, \dots, b, c$  constants. This transformation is performed by simplifying **case** expressions, transforming the expression into the conjunctive normal form (CNF) and extracting the clauses.

To simplify **case** expressions, observe that an expression of the form **case**  $c_1 : a_1 ; c_2 : a_2 ; \dots$  **otherwise**  $a_n$  can be replaced by the (conjoined) implications  $(c_1 \rightarrow a_1) \wedge (\neg c_1 \wedge c_2 \rightarrow a_2) \wedge \dots \wedge (\neg c_1 \wedge \neg c_2 \wedge \dots \wedge \neg c_{n-1} \rightarrow a_n)$ . Starting from the bottom of the tree, all Boolean case expressions can be replaced in such a manner.

In the event of nested **case** expressions, the Cartesian product of the conditions will have to be taken, as a Boolean expression cannot be nested in an expression of numeric type.

The resulting expression is changed into the CNF (pushing negations inwards), with inequations normalized into  $c = a \cdot x + \dots + b \cdot y$ . As a final step, expressions are simplified, for example by folding constant expressions.

The SLIM model is then serialized into a format that is based on the Ecore representation of AADL [AS5506-1]. This is an XML-based format that is easier machine-parseable than the human readable models using the syntax from Chapter 2, which is described further in Section 5.2.3.

### *Properties*

The simulator supports properties specified in CSL without nested probabilistic operators. Similar to the SLIM input, an XML-based format is used, though its structure is specific to COMPASS.

Properties are of the form  $\text{Pr}_{\approx}^?[[]\varphi]$ , meaning it queries for the probability of the formula  $\varphi$ . Here,  $\varphi$  is the time-bounded until operator (i.e.,  $\varphi = \psi_1 \mathcal{U}_{[l,u]} \psi_2$ ) or an operator that can be derived from it (such as  $\diamond$ ). The upper bound  $u \in \mathbb{R}$  is used to determine the maximum simulated time.

#### 4.6.2 MODEL REALIZATION

After reading in the SLIM model, its *realization* is constructed, which is used to generate the EDAs and NEDA that will be simulated. The process itself is fairly straightforward, and functions in the same manner as for other aspects of the COMPASS toolset. However, a few concerns have to be taken into account.

### *Expressions*

As the majority of the time in the simulator is spent on calculating the next state, emphasis was put on the performance of expression evaluation, which occurs during checking guards and invariants. For each known operator and type in the SLIM language, a separate function is generated by the compiler. As the simulator is written in the C++ language, templates are used for this purpose. This allows specialized functions to be generated for all the operators, reducing the overhead of evaluating expressions to a minimum. Further reductions would require the expression to be compiled to machine code directly.

As previously mentioned, guards and invariants are also transformed into clauses. These are used for the specific purpose of determining the time bounds in which they evaluate to true (if any). For each clause, the interval is determined in which the contained (in)equality is satisfied. As they are always of the form  $c \bowtie a \cdot x + b \cdot y \dots$ , where  $a, b$  are determined by the change rate specified for the current state—parameter  $\phi$  in terms of an EDA—and  $c$  depends on the current valuation for

$x, y$ , this interval can easily be determined. The intersection is then taken for each determined interval of all clauses to determine the complete, possibly non-convex, interval of the guard or invariant in which it evaluates to true.

### *Event and Data Graphs*

When determining the next discrete state of the system, a decision has to be made about which transitions will be taken next. As the simulator uses an explicit representation of the system, it needs to determine which transitions, if any, synchronize on the transition scheduled to be taken next. For the NEDA, the relation  $EC$  represents the event connections that are relevant. However, it is defined as a mapping from each possible combinations of modes to connected events. Due to the fact that the number of such combinations is potentially very large, it is impractical to store all possible combinations of connections in advance. Instead, the complete graph of all possible connections is constructed in advance. Edges in the graph are annotated with the condition in which they can be active: Both the connection and the EDA it is contained in should be enabled in the current mode. As the discrete state is updated, the edges of the graph will follow the state of active connections, though the topology will not have to change.

A similar approach is taken for data flow graphs. However, for data flows the order in which they are evaluated is relevant as well. Therefore, when the graph is built, the edges are ordered according to their dependencies. SLIM does not permit for cyclic data dependencies, which guarantees this is possible. When updating the discrete state, the flows are then evaluated according to their ordering.

### 4.6.3 SIMULATION

The simulation engine consists of two parts: a *generator* that keeps track of the generated samples and determines if more are necessary, and a *discrete event simulator* that generates the samples. The discrete event simulator in turn is based on a simulation strategy, a NEDA (realized model) and a property.

A generator accepts the value 1 or 0 for each generated trace, depending on whether the property under investigation holds or not, and indicates whether or not more samples are needed. In the case of SLIMSIM, a single generator is supported for quantitative analysis, which is based on the CH-bound. That is, based on input parameters  $\epsilon$  and  $\alpha$ , it determines a fixed number of samples a priori. Thus, it will indicate further samples are required until the CH-bound has been reached (or exceeded).

The discrete event simulator generates the actual traces of the model. This is done iteratively: A loop generates steps throughout the model, until the property can be shown to either hold or not. Which step is to be taken next is determined by the active strategy as described in Section 4.5.

Strategies in the simulator act as a scheduler for the non-deterministic system. As such, they draw random samples to determine what delays to apply for timed

transitions, and which discrete transitions are taken. Timed and discrete transitions are always taken in an alternating fashion. In particular, this allows the strategy to associate random delays with a specific probabilistic event, ensuring discrete transitions are taken to the exact probabilistic distribution association with them.

For random number generation, the SFMT [122] library is used. It provides an implementation of the Mersenne-Twister pseudo random number generator (PRNG), which is suitable for statistical applications.

The evolution of the state is determined by the NEDA. Given the transitions to be taken by the strategy, the NEDA updates the state variables. Local and port variables are updated according to the data flows that are in effect, and the effects associated with the taken transitions. These variables are double buffered to ensure updates only depend on the current state, and not the next. Following, the next mode is determined for each individual EDA, including the activation of possibly deactivated EDAs. Finally, the event connection and data flow graphs are made up-to-date.

### *Properties*

The outcome of properties are used as the primary termination condition for the generated traces. To make this possible, the evaluation of a property results in either *true*, *false* or *unknown*. The latter is the result when the property contains path operators, for which (future) states may have to be determined before it can conclusively said to be true or false. As long as the property evaluates to unknown, more steps are generated.

#### 4.6.4 DESIGN CHOICES

The design of the simulation algorithm of SLIMSIM was made with the specific semantics of SLIM in mind, as well as the goal to provide a performability approach for COMPASS.

Discrete and timed transitions are taken in tandem, with the order between the two being determined by the selected strategy. The delay is allowed to be zero, permitting effectively multiple discrete transitions in succession, but not timed. This is in correspondence with the semantics of SLIM. Non-deterministic choices in SLIM models are treated as equiprobable alternatives, for the same reasons as mentioned in Section 4.3.2 (in fact, all known SMC tools take this approach, or reject such models).

Process scheduling is done by choosing uniformly between the processes with the shortest delay. Unused delays based on clocks are retained unless the configuration of the system changes, to avoid introducing a bias depending on the strategy selected. Those based on the exponential distribution (for error events) are resampled. Due to the memorylessness property of this distribution, this does not introduce a bias.

TABLE 4.3 – Overview of the different approaches taken by SLIMSIM, UPPAAL(-SMC) and MODES.

Tool	Order	Accuracy	Scope	Policy
SLIMSIM	Delayed	Strategy dependent	Strategy dependent	Mixed
UPPAAL	Delayed	Approximate	Local	Resampling
MODES	Delayed	Approximate	Local	Age Memory

Deadlocks can either be treated as errors, or alternatively be considered a state in which the property under evaluation will never become true. Zeno behavior is not actively detected as this is not tractable in the current implementation, rather a (configurable) maximum number of steps is defined.

A comparison can be made with the UPPAAL(-SMC) and MODES tools. These were selected from those mentioned in Section 4.2.3 as they support the use of continuous distributions and, to some degree, non-determinism (MODES by default disallows non-determinism, but this can be overridden). A more detailed explanation of how these aspects were determined can be found in [HB1]. Table 4.3 shows how these tools can be classified according to the categories from Section 4.4.

These choices have a measurable effect on the probabilities returned by the tools depending on the input model. Referring back to some of the examples presented in Section 4.3, each tool can result in a different reachability property for the various locations in the automata presented (encoded in the respective language of each tool). The expected probability of some reachability properties is shown in Table 4.4, including the effects of the strategies provided by SLIMSIM.

## 4.7 CASE STUDY

The SLIMSIM simulator was tested with a case study representing the design of a launcher. The case study was designed to test the simulator as well as the use of timed variables in SLIM models with a probabilistic nature. It models a hypothetical *launcher* — not based on a real design — which permitted some freedom in the choice of certain modeling parameters.

A launcher is generally short-lived. Its primary task is to bring its payload into orbit, meaning its lifespan is in the order of a few hours. However, as one might imagine, a very high availability is required, as the loss of a critical components for a few milliseconds can cause the loss of the launcher and its payload entirely.

One method of ensuring the required amount of availability is to run critical systems in a redundant configuration. The case study employs two methods: *warm* and *hot* redundancy. The prior runs two or more systems, of which only one communicates with its environment. In case of failure, control is switched to one of the redundant systems instead. The latter again has multiple systems running. However, the outputs of all of these systems are considered, in the case study by a *voter*.

TABLE 4.4 – Effect of strategies on expected reachability probabilities for the tools UPPAAL, MODES and SLIMSIM, based on the examples in Section 4.3.

Example	Property	Probability as determined by:			
		UPPAAL	MODES	SLIMSIM	
Figure 4.2	$\Pr_{p=?}[(\diamond^{[0,4]} p_1)]$	0.75	1.0	ASAP:	1.0
				Progressive:	0.75
				Local:	0.75
				MaxTime:	0.0
Figure 4.4, P	$\Pr_{p=?}[(\diamond^{[0,5]} p_1)]$	0.8	1.0	ASAP:	1.0
				Progressive:	0.8
				Local:	0.8
				MaxTime:	0.0
Figure 4.4, Q	$\Pr_{p=?}[(\diamond^{[0,5]} q_1)]$	0.2	1.0	ASAP:	1.0
				Progressive:	0.5
				Local:	0.2
				MaxTime:	0.0
Figure 4.4, R	$\Pr_{p=?}[(\diamond^{[0,5]} r_1)]$	0.9	1.0	ASAP:	1.0
				Progressive:	0.5
				Local:	0.9
				MaxTime:	0.5

The voter compares the output of the various systems, and votes on which systems outputs correct information in case there is a discrepancy.

In case a component of a redundant system has failed, multiple approaches can be taken to mitigate this. The simplest option is to disable the defective component, and resort to a redundant one. This approach is taken in the current generation of launchers, such as the Ariane 5. More sophisticated approaches will try to bring the failed system back under nominal conditions, either by waiting, or by resetting it—possibly by means of power cycling. It may happen that either the failure condition remains, at which point the component is considered permanently defective, or the issue may be resolved. In the latter case, the component may be further monitored more closely, and considered permanently failed if failure occurs again.

Allowing a system to recover a component that failed in the past can improve availability. In case some other component fails, the recovered component can take over, preventing a complete failure of the system itself. This approach is considered for future generations of launchers. However, due to the extra complexity required to implement this, ensuring proper verification of the system is imperative.

In the case study, both the possible error conditions and the recovery possibilities were taken into account. It takes some modeling strategies into consideration, and permits verifying the statistical model checker is capable of handling such models.

#### 4.7.1 NOMINAL MODEL

The system being modeled is a launcher consisting of four groups of subsystems:

- » Power conditioning and distribution;
- » navigational input;
- » data processing;
- » the thrusters.

The launcher itself is modeled with an `offMode` mode, various initialization modes and an `onMode` mode. The model starts in the `offMode`, and transitions via the initialization modes to the `onMode`. This allows for a warm-up in which not every system is in its nominal condition. Verification can then be performed such that properties check that everything is nominal in the `onMode` mode.

The power of the launcher's systems is handled via power conditioning and distribution units (PCDUs). Each PCPU contains a battery to provide power, and some power outputs. The battery has a charge level associated modeled using a continuous variable, linearly decreasing the charge level as time passes.

The navigation is modeled by means redundant GPS and gyroscopic (Gyro) sensors, allowing both position and attitude to be measured. Compared to the GPS device described in Chapter 2, the one used in the case study is simplified to have two modes: `acquisition` and `active`, with an unconditional transition from the first to the second with some time delay (between ten seconds to two minutes). Two GPS devices run in parallel, with the system state considered nominal during launch.

The gyroscopes are modeled as devices containing two sensors aligned perpendicular to each other. The whole launcher model contains three such gyroscopes, providing two sensors for each axis. The system is then considered nominal as long as one sensor per axis is working. Each sensor holds a Boolean variable to indicate if it is working.

The input from the navigational devices is processed by two on-board computers (OBCs), which contain three data processing units (DPUs). The DPUs are responsible for processing the navigational data in order to send the right commands to the thrusters. In each OBC, the three DPUs are part of a voting triplex, allowing two DPUs to fail before it itself is considered to have failed.

To model the power distribution, and effect of loss of power, each device is modeled as a system containing a subcomponent that implements the actual behavior of the device in question. The system has two modes, a powered mode in which the subcomponent is enabled, and an unpowered mode in which it is disabled. This approach simplifies the modeling compared to avoiding this indirection; when embedded directly, the containing system would have to specify the reconfiguration for all combinations of enabled and disabled redundant components directly. Furthermore, the approach taken also permits the effect of power loss to be modeled solely by the component itself, separating it from the behavior of the system.

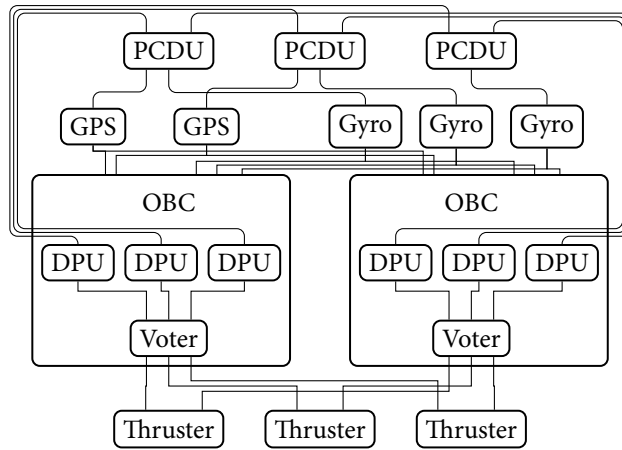


FIGURE 4.7 – The architecture of the industrial case study. The connections between the GPS, Gyro and DPU units have been hidden for clarity. Rounded connections are for power, the others for signals.

Power is distributed such that each PCDU powers one gyroscopic device, and one DPU from each OBC each. Both GPS units draw power from a separate PCDU each. As such, the existence of a single point of failure is avoided.

Finally, three thrusters are modeled. Each thruster is considered operational as long as at least one OBC provides correct commands to it. Failure of a thruster would mean the entire system has failed.

#### 4.7.2 ERROR MODEL

The case study considers three types of faults in the system: *transient*, *hot* and *permanent* faults. They are defined as in Section 2.3. The rates for these faults are chosen arbitrarily, though with transient faults having the highest rates, and permanent faults the lowest. The transient faults recover with a non-deterministic time delay. This delay falls in the range 200–300 ms.

Fault injections are specified to induce errors either in the output power or output signals. In case of any type of fault, the power is disabled for the PCDUs, the sensor signal disabled for the GPS receivers and gyroscopes, and the command signal from the DPUs.

#### 4.7.3 EXPERIMENTAL RESULTS

Some experiments were run to see how the possible configuration of the simulator would affect the outcome. To this end, two versions of the model were used: One where repair of the DPU was not possible, and one where it was, with the success

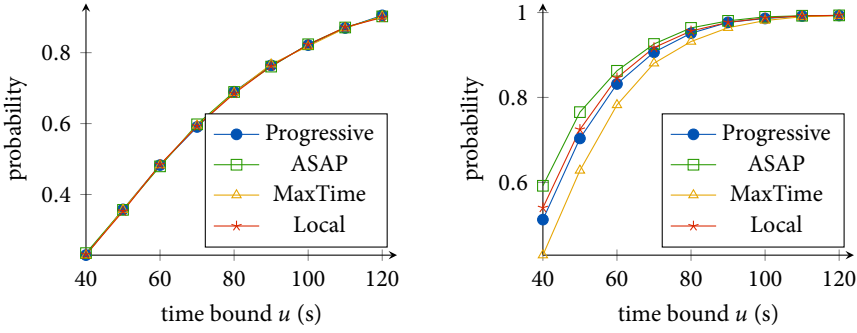


FIGURE 4.8 – Probability of system failure for both benchmarks. On the left repair was not possible, on the right it was.

rate depending on the time of repair. Then, a single property was checked under each configuration and strategy.

The property under investigation was specified as  $\Pr[\diamond^{[0,u]} failure]$ , where  $failure := mode = onMode \wedge \neg triplex1.cmd \wedge \neg triplex2.cmd$ . This matches the probabilistic existence pattern as described in Chapter 3. The property checks the probability of complete failure (by means of both triplexes in the OBCs failing) within timebound  $u$ . By varying the time bound, some insight can be obtained about the development of the failure probability over time.

The property was checked for two variants of the system: One variant does not permit recovery of a DPU that had temporarily failed. The other variant does, by resetting a failed DPU and waiting a fixed amount of time before checking if it has recovered. The error model has been adjusted such that a reset of the device too early while it has failed, will cause it to fail permanently. This is done to highlight the differences between the various strategies of the simulator.

Experiments were run with the CH-bound parameters  $\alpha = 0.9$  and  $\epsilon = 0.05$ . The results are displayed in Figure 4.8. The plots for the system without repair all show values within the same confidence interval. The reason is that the model does not contain any underspecification, thus the strategies have no influence. In the case of the system with repair, in which time bounds were introduced, the effect of strategies becomes visible. The ASAP strategy shows the highest failure rate. This is due to the fact that the model causes a DPU to fail permanently if it gets reset too early, which the ASAP scheduler will always do. On the opposite end is MaxTime, which will never do so, this allowing to recover the failed DPU. The other two strategies lie in between. Here, the Local strategy shows a slightly higher failure rate than the Progressive one. This is due to the fact that it makes it more likely a delay is chosen for which the nominal model cannot take a transition, allowing the error model to preempt it.

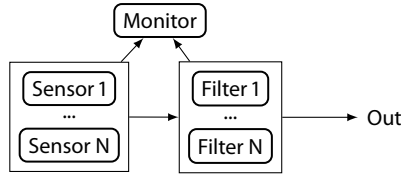


FIGURE 4.9 – The layout of the sensor-filter model.

#### 4.8 COMPARISON WITH PERFORMABILITY

Prior to the implementation of SLIMSIM, the COMPASS toolset supported the analysis of probabilistic models only by means of CTMC and IMC model checking. For this, the MRMC and interactive Markov chain analyzer (IMCA) tools were used. Although SLIMSIM does not *replace* this approach, it can be used as an alternative. Some benchmarking was performed to get insight in the difference in performance characteristics between the two approaches.

The benchmark compares the approaches for CTMC models, both because CTMC model checking in COMPASS has a better performance than IMC model checking, and non-determinism is absent so the resulting outcomes can more easily be compared. COMPASS requires that for CTMC analysis the entire state space is generated. This involves multiple steps and tools, as will be discussed in Section 5.2.3. Because of this, the expectation is that simulation can generate results much faster by avoiding the costs associated with these steps.

The model being benchmarked is based on the *sensor-filter* benchmark included in the COMPASS toolset. The model represents a system which consists of  $N$  sensors and just as many filter components. A sensor produces values in some expected range. A filter is a function that processes this value and produces another one. A monitoring component compares the values from both type of components against the expected ranges. If the value lies outside this range, the component is considered failed and a signal is given to switch to the next redundant component, up to  $N$  times. If either  $N$  sensors or  $N$  filters have failed, the system itself fails. An overview of the structure is shown in Figure 4.9. Error models are associated with each sensor and filter component, with fault injections causing values to go out of range. The error models contain a simple transition from a nominal to error state, with a fixed error rate. The entire model is fully probabilistic without non-determinism (otherwise, CTMC analysis would not be possible).

A time-bounded property was specified that checks for the existence of system failure. The parameters used for simulation were  $\alpha = 0.98$  and  $\epsilon = 10^{-3}$  and  $\epsilon = 10^{-4}$ , and twenty parallel threads. MRMC was configured to use a precision of  $10^{-6}$ .

The results are shown in Table 4.5, measured on an Opteron 6172 @ 2.10 GHz system. As expected, as the model gets larger, and thus its state space, the model-checking time using CTMCs grows exponentially. The simulation time increases

TABLE 4.5 – Benchmarking results for the sensor-filter model. The size indicates the number of both sensors and filters. The values for the simulator are maxima.

Size	States	CTMC Time (s)	Simulation ( $\epsilon = 10^{-3}$ ) Time (s)	Simulation ( $\epsilon = 10^{-4}$ ) Time (s)	CTMC Memory (MiB)	Simulation Memory (MiB)
2	192	5.33	47.50	4 453.09	23.19	19.91
4	768	29.93	51.61	4 887.09	76.29	21.99
6	1 728	59.75	50.58	4 725.42	89.14	24.21
8	3 072	289.97	52.65	4 801.40	180.22	26.63
10	4 800	725.67	52.65	5 120.87	178.35	29.13
12	6 912	1 360.11	57.80	5 398.62	196.43	32.75
14	9 408	3 187.02	59.88	5 668.64	2 469.20	35.00

slowly in comparison. However, reducing the size of the error bound by a factor of 10 increases the simulation time by a factor 100 (as expected according to the complexity metrics of the CH-bound). Memory usage<sup>3</sup> for the simulator increases slowly. The memory consumption consists mostly of the model description, which only increases slightly as  $N$  is increased. The state space, however, increases dramatically, and likewise the amount of memory required for CTMC analysis, as it must be stored in its entirety.

Although from the example it is clear the CTMC-based analysis can easily consume more time and memory as the model grows, simulation does not provide a “silver bullet” (as highlighted in Section 4.3). The CTMC-based approach gives a great deal more accuracy and precision, in particular when it comes to events with low probability.

## 4.9 DISCUSSION

The application of SMC within the COMPASS toolset was investigated to provide support for probabilistic analysis for models employing the full semantics capabilities for SLIM, including the hybrid aspects. As the numerical-based approaches did not support this, the SMC -based approach was chosen.

The Monte-Carlo method provides as tractable approach, powerful enough to support any SLIM model. This does not come without caveats however. Some scheduler has to be applied to a non-deterministic model, for which it cannot be said how it will affect the probability directly, though it is assured to be bounded by the theoretical minimum and maximum.

Although SLIMSIM fulfills the needs it was originally designed for, some points remain where it could be improved to make it more efficient or provide a wider range of supported input or results.

<sup>3</sup>Measured using the resident set size (RSS).

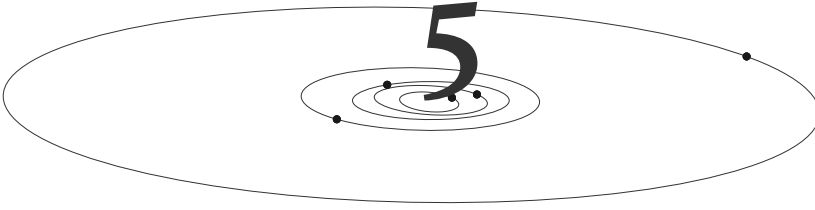
Currently, SLIMSIM only provides quantitative results, and uses the CH-bound to generate them. However, often the exact probability is not required, the knowledge if it lies below or above some threshold is usually sufficient. For this, support for hypothesis testing needs to be added, ideally with a selection of algorithms to provide a close match with the characteristics of the property under investigation.

On the quantitative side, allowing the estimated probability distribution over time to be plotted would be helpful as well. The CTMC and IMC-based approaches support this, SMC does not.

Support for importance splitting (cf. Section 4.3.9) could be added as well. In particular for systems with a high criticality, the detection of rare events, or confidence in their absence, becomes crucial. Though numerical methods are more suited to the task, it is a very valuable feature for an SMC -based approach, in particular when no alternative is possible.

Finally, supporting a different input model would make the tool more versatile and maintenance friendly. COMPASS\* (see Chapter 5) considers supporting a larger class of input models. In addition, the current model realization step used by SLIMSIM duplicates some of the work already supported by the COMPASS architecture.

When both numerical analysis and SMC are possible, SMC may be able to produce results faster. However, for probabilities close to 0 or 1, numerical analysis can provide better performance. Usually, the choice between the two involves a trade-off in performance and quality of the result. The primary advantage of SMC is making the analysis of complex models tractable. However, its caveats, in particular with respect to scheduling, should be taken into account.



## THE COMPASS TOOLSET

*ABSTRACT – This chapter describes the functionality and architecture of the COMPASS toolset. A global overview of the available analyses and the approach taken to implement them is given, as well as the overall architecture detailing the interaction of the various tools in the toolset.*



**J**OINING theory and practice, the COMPASS project<sup>1</sup> is an international research project aiming to ensure system-level correctness, safety, dependability and performability of on-board computer-based aerospace systems. As part of this project, the COMPASS toolset was created, which provides an integrated approach to implement these goals.

The toolset is designed to perform various orthogonal analysis techniques from a single input formalism. This avoids the need to provide multiple specifications by hand, as is common practice even today. COMPASS allows the same model to be used for, e.g., correctness reliability, and safety analysis, providing artifacts such as fault trees (cf. Section 5.3.6), timed failure propagation graphs (TFPGs) (cf. Section 5.3.9) and counterexamples (cf. Section 5.3.3). Adjusting the model only requires using the toolset again to regenerate these artifacts. This can be done automatically, preventing them from becoming inconsistent.

The particular version of COMPASS being discussed is COMPASS 3.0, the latest iteration of the toolset at the time of writing, which was released in 2017. It is the result of a consolidation effort of the results of various follow-up projects. Since its first public release, COMPASS has been extended and improved upon, most notably adding new features that were developed as part of various projects that used it as a basis. These projects, and their goals, are discussed in more detail below.

---

<sup>1</sup>Short for Correctness, Modeling and Performance of Aerospace Systems.

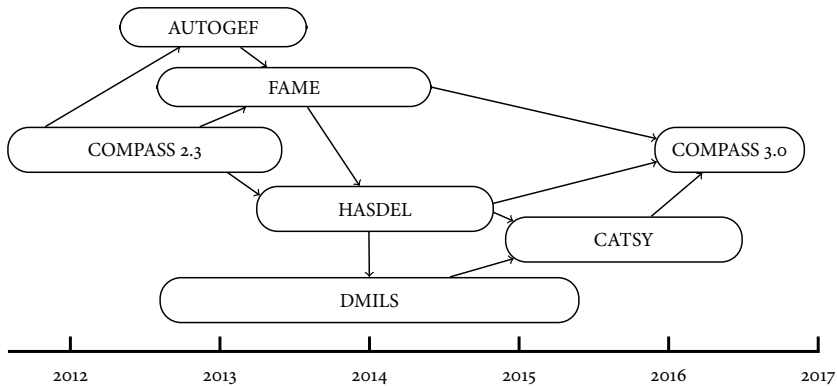


FIGURE 5.1 – Overview of the relation between the various COMPASS projects. Arrows indicate the dependencies between the various projects.

## 5.1 (SUB)PROJECTS

The current version of COMPASS is the result of research and extensions of various spin-off projects that were started after the inception of COMPASS itself [50]. Since its first public release, various new features have been added that were developed in these projects, finally integrated as part of the COMPASS 3 project. Figure 5.1 shows the relation between these various projects, with the more recent projects toward the right. The following will give a brief description for each project.

**AUTOGEF** The first project following COMPASS was AUTOGEF (Automated Model Generation Toolset for FDIR), which had the primary goal of automatically synthesizing FDIR specifications based on dependability requirements of a space project [3]. More concretely, based on a SLIM model and a set of unwanted system behaviors generated using COMPASS safety assessment analyses (cf. Section 5.3.6), an FDIR specification in SLIM is generated.

In addition to this functionality, the mission specification feature was added as well, which provides the possibility to define various phases and operational modes that apply to the specification, and bind them to concrete SLIM modes and properties. This specification can then be used as an additional input for the FDIR synthesis.

Though the synthesis functionality has not been backported to COMPASS, the mission specification feature (cf. Section 5.3.2) has been integrated into COMPASS 3.0.

**FAME** The FAME (failure and anomaly management engineering) project [71] followed AUTOGEF, and aimed to improve the overall FDIR support. In particular, support for the analysis of timed failure propagation models (TFPMs) was added. This has been achieved by implementing support for the synthesis and analysis of TFPGs as will be described in Section 5.3.9.

**HASDEL** HASDEL (short for Hardware Software Dependability for Launchers) was a project aimed to improve the analysis capabilities for launchers, in particular with respect to advanced fault-management and timed analysis.

HASDEL added better support for specification of timed behavior in SLIM models, in the form of timed error models, and the introduction of time units and time delay specifications. It also introduced the syntax for reactivation transitions.

On the analysis side, correctness checks were added for Zeno behavior and time-divergence, discussed in Section 5.3.3. Diagnosability was extended to support the specification of diagnosability delays. Performability analysis was improved by providing Monte-Carlo simulation based techniques, which are discussed in detail in Chapter 4.

**D-MILS** The D-MILS (distributed MILS) project, as an exception to the other projects, did not solely focus on an implementation based on COMPASS, but rather integrated COMPASS into a new distributed platform based on multiple independent levels of security (MILS). The project had a strong focus on the analysis of distributed systems, in particular with respect to information security between components, as well as leveraging assurances from individual components to provide them for the distributed system.

Based on these goals, the toolset was extended in two ways:

- » Extension of SLIM to support a type system that permits reasoning over the security of data;
- » Support for CBA, both by extending SLIM to support the specification of contracts, and adding the analysis capabilities by introducing new tools in the back-end.

The security-aware type system made it possible to annotate the data in the model with a privilege level. High and low-level privilege was supported, though theoretically more levels can be supported as well. This makes it possible to verify that no information leakage of high-level data is possible via low-level channels [118].

CBA added support for compositional analysis. Syntactically, SLIM was extended to support so-called annotations, a precursor to the current AADL properties. These annotations were used to specify contracts for individual components. Tooling was added to support their analysis as well, in particular a backend for the OCRA tool. This functionality was later extended in the CATSY project.

**CATSY** The main goal of the CATSY project was to improve early validation and verification activities. As a result, a requirements taxonomy and catalogue of system and software properties (CSSP) were created, which have been described in Chapter 3 (with technical details in Appendix C).

The toolset has been extended by improving the possibilities for property specification, the specification of fault injections and improvement of contract-based

reasoning. The latter includes the support for generating hierarchical fault trees, discussed in Section 5.3.6.

In addition, the support for CBA, introduced in the FAME project, has been improved, and updated to reflect a syntax based on AADL properties. Tool support was extended to support a wider class of models, in particular those which specify modes in composite components, which was not possible prior.

**COMPASS 3** In order to consolidate the results of the prior mentioned projects, the COMPASS 3 project was started with the aim to provide version 3.0 of the toolset, incorporating the advancements of past projects. Furthermore, the SLIM language was brought further in line with the AADL language, improving compatibility with existing AADL tooling. In addition, an overall cleanup of the code base was performed as well, to be in line with current programming practices. The functionality described in this chapter is based on the final result of this project, as is the version of SLIM as described in Chapter 2.

## 5.2 ARCHITECTURE

The COMPASS toolset architecture consists of a front-end, written in Python, and various back-end tools. The front-end is responsible for reading the user input and specifications, transforming them such that they can be used by the back-end tools, invoking those tools, and parsing their input to present them to the user. The back-end tools provide the core analysis facilities and are written in either C or C++.

Figure 5.2 shows an overview of the architecture. The user provides the SLIM model and the possible property types. Using either the GUI or command line interface (CLI), any of the possible analysis tasks are invoked. This starts with compiling the SLIM model and properties into the internal representation used by the toolset. Afterward, this data is translated into the formats supported by the analysis tools. The analysis tool(s) are then invoked, and their results are presented to the user.

### 5.2.1 MODELS

The SLIM models are provided using the syntax (and with the semantics) as described in Chapter 2. They can be passed via the compiler directly to the translators for the back-end tools, however, the model extension (cf. Section 2.4) component can be called to transform the model to inject error models. When reading in a SLIM specification, the toolset first parses it into Python data structures (using ANTLR 3), which are subsequently used throughout the toolset.

One important aspect of compilation is that an *instance* of the input specification has to be created, which is based on a *root component*. The root component is the topmost component in the component hierarchy, and can be selected by the user (though for ease of use, if a single component implementation exists that is not a subcomponent of another it will be selected as the root component by default).

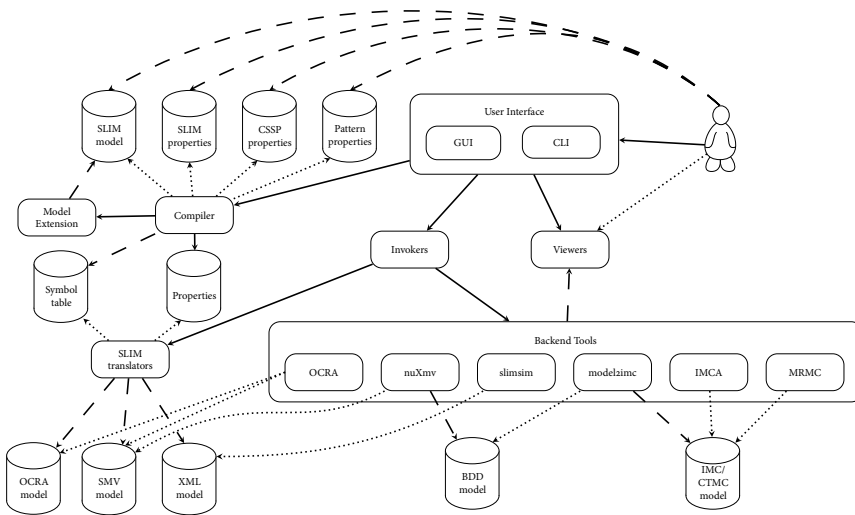


FIGURE 5.2 – Overview of the COMPASS toolset architecture. Dashed arrows indicate where data is provided from, dotted arrows a dependency on data. Straight arrows indicate which component invokes another.

From the root component, subcomponents are instantiated according to the component implementation (or type) specified in the SLIM model for that subcomponent. Due to the application of AADL properties, variations can exist between instances of the same implementation, most notably due to model extension. By default, the root component is selected automatically when there is a unique implementation that does not occur as a subcomponent of another. If no such implementation exists, the user is asked to provide the root component. After the model has been instantiated, henceforth referred to as the *instance model*, it can be used for subsequent analyses.

Depending on the data types used in the model, it can be classified in different categories which will influence the possible analyses that can be performed on it. First are *discrete* models, which can either be of *finite* or *infinite* size. These models contain discrete types, which are either countable finite (such as **enum** or  $[l..u]$ ) or countable infinite (such as **int**).

When using uncountable infinite types such as **real**, the model becomes *continuous*. If in addition the time domain is considered by the use of either **clock** or **continuous** variables, the model is said to be *timed*.

### 5.2.2 PROPERTIES

One of the key inputs behind COMPASS analysis of models are the properties that can be specified, specifically formal specifications that encode certain behavior in the model. Such properties can be formal representations of requirements, as described in Chapter 3, but also encoding of particular states, such as *error states* which the system should avoid and recover from.

The diagram in Figure 5.2 shows the three different types of properties, which are either those specified in the SLIM property syntax (see section A.6), those based on the CSSP (see Section 3.4), and those based on patterns (see Section 3.3.1). The compiler translates all these into a canonical internal representation, suitable to be translated into the formats of the various back-end tools.

One important aspect is that the properties in the SLIM model need to be *instantiated* in order to be applicable to the final compiled instance model. This means that for properties defined by component types and implementations, a corresponding instance needs to be created for each instance of these component types and implementations in the instance model. Each of these property instances then has to be individually verified. (though the user may decide to verify only a subset).

For `nuXmv`, both LTL and CTL are supported. In addition, certain timed properties based on patterns can be expressed, which map to MITL. For OCRA, a syntax is used that closely resembles that of SLIM property expressions. The `SLIMSIM` tool uses an XML-based format to represent CSL properties, which is derived directly from the internal representation used by COMPASS. The `MODEL2IMC` program uses its own representation, which encodes properties in Polish notation. For MRMC a translation for its CSL representation is used. For IMCA instead a fixed set of patterns can be verified (similar as the MITL patterns for `nuXmv`) by translating them into the corresponding program options for IMCA.

### 5.2.3 MODEL TRANSLATION

In order to make the SLIM model available to the various tools in a supported format, various translations are required. Such translations permit the various tools that comprise COMPASS to communicate.

**SMV** The primary backend format for COMPASS models is symbolic model verifier (SMV). This is the input format for the `nuXmv` [38] and `nuSMV` [47] model checkers. The SMV format provides a symbolic representation of the model, expressed using transition relations, invariants and input variables. SMV specifications consist of various modules that can communicate using variables, which are used to represent the components of the model and their port connections.

**OCRA** For CBA, the OCRA [44] tool is used (short for Othello Contracts Refinement Analysis), which uses its own input format, OCRA system specification

(OCRA). The OSS format consists of individual *components*, each specifying an *interface* and, optionally, a *refinement*. The interface defines the ports of a component, as well as any *contracts* (see Section 3.3.3).

The refinement specifies the subcomponents, and the connections between the subcomponents and the component itself. Furthermore, the refinement may specify further *contract refinements*. Briefly, contract refinements specify how contracts of subcomponents refine the contract of the component itself. More details are provided in Section 5.3.4.

SLIM models are mapped component wise, each component in the instance hierarchy mapped to one in OSS, with ports being mapped directly. SLIM subcomponents and port connections are placed in the refinement specification.

Finally, components in OSS can be associated with SMV modules, allowing the behavior of models to be included. The previously mentioned SMV translator is used for this purpose, however, modules for SLIM components are treated individually, using the OSS specification to build the hierarchy.

**Sigref/BDD** Probabilistic analysis as described in Section 5.3.5 requires the complete state space to be constructed, and subsequently made readable for either the MRMC [87] or IMCA [70] model checker. This is done in multiple steps. First, the SLIM model is translated to SMV. Next, a binary decision diagram (BDD) is generated by `NUXMV`. In addition, as SMV does not support the specification of probabilities, a mapping is generated from SLIM error rates to variables in the BDD.

The data format of the BDD, used to interchange the model between SMV and `MODEL2IMC`, is based on the one used by the `SIGREF` [141] library. It consists of a BDD encoding the initial state, a BDD encoding the transition relation, BDDs for the relevant atomic propositions (APs) (labels) that apply to the states in the transition system, and BDDs for the actions used in the transition relation, divided into the internal  $\tau$  transition, action transitions and probabilistic transitions. The latter is actually an algebraic decision diagram (ADD), with the terminals representing the rates by which the transitions occur. The probabilities are read from a separate file that maps the error event transition labels to their corresponding transition rates, which `MODEL2IMC` then injects into the terminals of the corresponding ADD.

The `SIGREF` library is used to minimize the state space induced by the given decision diagrams (DDs), which depends on the set of APs that apply. This set is based on the APs that occur in the properties of interest, meaning that these properties have to be selected a priori. The downside of this is that selecting a different property of interest may require a potentially expensive recalculation of the (minimized) state space. To alleviate this, `MODEL2IMC` can be instructed to construct a reduced state space based on the APs found in all properties, and store this. When analyzing individual properties, the stored state space can be loaded (and further reduced

if the set of APs in the selected properties is smaller) and transformed into the formats supported by MRMC and IMCA.

Both MRMC and IMCA use text-based formats to encode the models. MRMC stores transitions and labels in separate files [144], whereas IMCA uses a single file [69]. IMCA in addition encodes the *goal* states in this file, which are the state for which reachability properties can be determined. MRMC uses the state labels for this purpose.

**Sigref/DFT** For probabilistic fault tree analysis, the MRMC and IMCA tools are used as well, using SIGREF to generate their inputs. A similar approach is taken as for the BDD-based input, however, an extra step is required. Fault trees are provided as-is using the FT+ file format. This is an XML-based file format that describes the events and gates of a fault tree. SIGREF is used to parse this file and generate an IMC based on the events and gates of the fault tree. The approach taken is based on the work published in [29]. Each node in the fault tree is represented by an IMC. The various IMCs are composed using SIGREF, to minimize the resulting state space after each composition. The resulting final IMC is then passed on to the MRMC or IMCA for reachability analysis.

Each IMC generated for a node in the fault tree has a unique transition label associated with the transition that leads to the state in which the gate has been activated. For gates, transitions leading from the starting state are labeled the activation actions of their child gates. This allows for the necessary synchronization when composing the IMCs together.

The various states in the IMC are labeled according to the gates that have been triggered in that state. This makes it possible to define properties referring to these gates in their atomic propositions.

**XML/Ecore** For statistical analysis the SLIMSIM tool is used. The format used by SLIMSIM is a modified version of the Ecore-based format used to represent SLIM in a machine readable format, which in turn is an extension of the AADL variant of this format. Ecore lies at the basis of the Eclipse Modeling Framework (EMF) [58], a modeling framework based on a structured data model, Ecore. Ecore models can be represented in XMI, in the case of AADL as described in [AS5506-1].

SLIM extends this model to support features that cannot be found in AADL, which are listed in Section 2.6. For SLIMSIM, support is added to store the clauses of guards and invariants as discussed in Section 4.6.1.

This format can also be used as a machine readable interface format for SLIM models. This is in particular useful for the AADL-compliant subset, as Ecore permits data-interchange between different tools.

### External Tools

COMPASS employs some further data formats that are used by external tools (those that are not directly integrated into the toolset), but may still be used with models that are based on SLIM.

**Simulink** A well-known modeling tool in the industry is Simulink [126], which is part of the MATLAB environment. Simulink permits a model to be constructed from basic blocks, supporting either discrete or continuous signals. One major advantage of Simulink is that it supports generating code for a multitude of platforms, in particular those that are embedded.

In Chapter 6 it is shown that SLIM models can be translated into Simulink specifications, which can subsequently be used to generate code, so that the benefits from both modeling paradigms are combined. For this, a translation of SLIM to Simulink was developed.

**BIP** Another modeling paradigm can be found in the behavior, interaction, priority (BIP) framework. Similar to SLIM and AADL, it supports model consisting of atomic and composite components that can be connected. The support to translate SLIM into the BIP-language was added as part of the distributed MILS (MILS) project [54]. The case study in Chapter 7 was originally written in BIP as well, though in that case the model is translated *to* SLIM.

**SXML** A graphical modeling tool for COMPASS can be found in the COMPASS Graphical Modeler (COMPASS). This tool can be used to display and edit SLIM models using a GUI, though limited to those using the syntax from the COMPASS 2 releases. The format used between this tool and COMPASS is the SXML format, which is an XML-based format designed by Ellidiss [57]. It is based on a more general AADL-based format, which is used by the AADL INSPECTOR tool [1], however, adapted to support SLIM specifically.

#### 5.2.4 BACK-END TOOLS

The various function of the COMPASS toolset are handled by a specific back-end tool, or a chain of tools. For each tool, a brief description of its features will be given. COMPASS hides most of the configuration required to use these tools. However, some options have been left for the user to control how the tool is used. These are described here as well.

#### *nuXmv*

NUXMV is the primary tool for generating models from SLIM, and checking its properties. It reads the SMV format as described above. It offers mainly those features pertaining to *correctness analysis*, as described in Section 5.3.3, and some of those offered for *FDIR analysis*.

Most importantly, `NUXMV` has three modes of operation that encode the model in different ways. For finite-state models, a BDD-based representation can be used, which provides a complete representation of the state space in a symbolic fashion. This permits the tool to give conclusive answers about properties provided to it.

The second mode of operation uses bounded model checking (BMC) or simple bounded model checking (SBMC), which explores the state space by visiting states that are  $k$ -steps away from the initial state, with  $k$  increasing from zero to some predetermined upper bound that is given by the user. The advantage of this method is that it does not require the model to be finite-state, and supports real numbers. However, depending on the size of the model and the bound set for  $k$ , it may be that it will not be explored fully. SBMC can check for completeness, but is available only for discrete models and can be computationally expensive.

Finally,  $K$ -Liveness [48] can be used for liveness properties by encoding them as an invariant and constructing a monitor that checks an accepting condition, which needs only to occur a finite amount of times. It is based on the IC3 algorithm, which allow these properties to be checked with a comparatively high efficiency.

The choice of operational mode has a great impact on the analysis of formal properties, which for `NUXMV` can be encoded in LTL or CTL. Whereas for BDD-based analysis a property can always be proven to hold, for the other methods this is not the case. If a counterexample is found, the property is certainly false, but in absence thereof, the result may be unknown (unless the search was complete). Furthermore, for the bounded approaches a counterexample must contain a loop. Thus, in the presence of deadlocks or diverging traces (which never return to a previously visited state), a counterexample may not be found, even if one is to be expected. To mitigate this, some functions are available to look for such behavior, which are discussed in Section 5.3.3.

### *xSAP*

For safety analysis, the `xSAP` tool (the successor of `FSAP`) is used. Specifically, it provides functionality for fault tree generation, failure modes and effects analysis (FMEA), TFGP analysis, and diagnosability analysis. These functions are described in more detail in Sections 5.3.6, 5.3.8 and 5.3.9 respectively. `xSAP` makes use of the `NUXMV` engine, therefore offering similar options as those mentioned for `NUXMV` itself.

### *OCRA*

The `OCRA` tool is used for CBA of the specification. As mentioned before, it uses the `OSS` language for specification of the system hierarchy and contracts, and the `SMV` for the behavioral implementation. Like `xSAP`, `OCRA` uses the engine `NUXMV`, and therefore also provides similar options as those described for `NUXMV`.

### *slimsim*

For probabilistic analysis based on the Monte-Carlo method, the SLIMSIM tool is used, which is described in more detail in Section 4.6. It uses an input format based on the Ecore syntax. Various options are provided by SLIMSIM which are presented to the user. First, the desired error bound and confidence interval have to be provided (see Section 4.2.2 for details). Second, the strategy to resolve non-determinism has to be selected, which is any of the strategies discussed in Section 4.5.

### *model2imc*

The MODEL2IMC tool is responsible for converting a BDD representation of the model specification, or a dynamic fault tree (DFT) specified in FT+, to a Markov model suitable for IMCA or MRMC. In order to do so, it requires both the model (either from SLIM or a fault tree) *and* the properties under verification, as IMCA and MRMC support different sets of properties. These properties are also used by MODEL2IMC to simplify the model. To do this, it makes use of the SIGREF library to minimize the model by means of (weak) bisimulation minimization. Based on the propositions in the properties, this process can be tuned to provide smaller models. The minimized model is then exported to the format of either IMCA or MRMC. The choice is normally made automatically based on the model and the properties, though the user may choose to override this.

### *IMCA*

IMCA can analyze reachability properties for IMCs, as well as expected time and long-run average metrics. These are used for performability analysis as described in Section 5.3.5. As IMCA approaches its solution for reachability properties by value iteration, it can be configured by setting the error-bound. COMPASS will prove a default for this value, but it is user configurable as well.

### *MRMC*

For the analysis of CTMCs, the MRMC tool is used. Though IMCA can perform this function as well, MRMC is more efficient for this particular type of model. MRMC can perform analysis using either numerical methods or simulation, of which COMPASS only makes use of the prior.

## 5.2.5 USER INTERFACE

Both a GUI and a CLI are provided. This makes it possible to choose to work interactively in a graphical environment, or command-driven.

The GUI allows the user to load a SLIM model, define fault injections, and specify properties, which can subsequently be used for model extension and verification respectively. The various possible operations are categorized under various *activities* (e.g., correctness analysis and safety analysis) and are accessible when the necessary

input data is provided. Various options can be directly configured from the GUI, and multiple operations can run in parallel.

The CLI allows single tasks (types of analysis) to be performed directly, without user interaction, which is primarily intended to be used for automation. All input data has to be provided a priori, and the resulting output is either written to the console, or written to files on disk for later inspection. For some of these, such as traces or fault trees (which are discussed in Section 5.3), the same visualization tools can be used as used by the GUI.

## 5.3 FUNCTIONS

COMPASS aims to provide an *integrated* approach to generate various artifacts from a single input model, all formally proven and used to validate and verify the model.

This section describes the various functions that the toolset has to offer, also including those that pertain to property specification (which aids the user in providing the input specification, rather than providing analysis output).

### 5.3.1 PROPERTY SPECIFICATION

Although properties can be specified by hand in the SLIM specification directly, this can be a rather cumbersome process. Instead, the GUI offers various interfaces to assist with the specification of properties interactively.

First, properties can be specified using patterns, using the CSSP or simply using SLIM logical expression as will be explained in the following. These specifications then get instantiated for the currently active model, and subsequently translated into the formalisms (i.e., formal properties) for the various analysis back-ends.

#### *Property Specifications and Requirements*

Perhaps one of the most challenging tasks when defining the specification of the system is the specification of the requirements, in particular the formal representations thereof. This problem has been discussed in Chapter 3, and COMPASS offers various approaches to assist the user.

From the point of view of the toolset, requirements can define either parts of the SLIM specification, or some property. The prior will have to be modeled directly, whereas the latter requires some other method of formalization that maps to a formal property which can be analyzed later.

COMPASS allows properties to be specified in any of the following three ways: using the CSSP, using a pattern, or specifying them as a SLIM logical expression. The CSSP has been introduced in Section 3.4, and makes use of AADL properties applied to the SLIM specification. Pattern properties are discussed in Section 3.3.1, and make use of placeholders in combination with a fixed grammar. Finally, SLIM

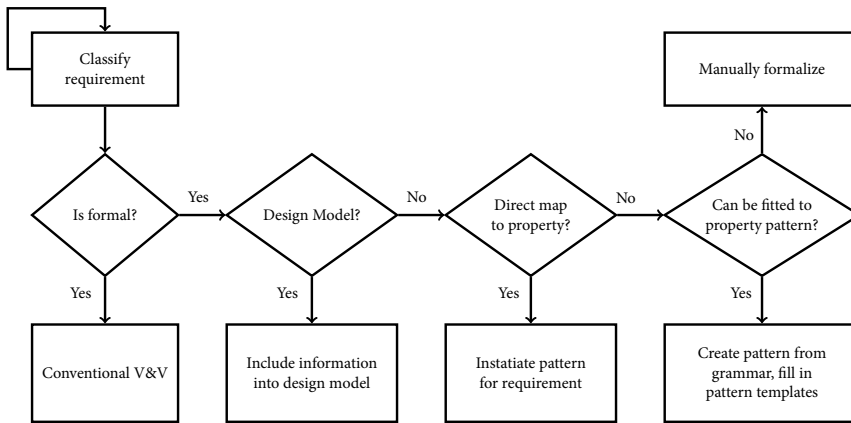


FIGURE 5.3 – Flow for selecting property specification approach.

logical expressions can be used to specify a formal property directly, using the operators that are given in section A.6.

Determining which approach to take to formalize a requirement can be done according to the flow depicted in Figure 5.3. As a first step, complex requirements need to be simplified. It can happen that a requirement is specified as a conjunction of two sub-requirements. Simplification then means that the requirement is broken up into its constituent parts, which may then be formalized.

Not all requirements are suitable for formalization (cf. Chapter 3). If this is the case, no further processing can be done by COMPASS, and other (conventional) V&V approaches should be applied. However, the requirement can still be added in COMPASS in plain text for traceability purposes.

If the requirement directly applies to the structure of the system, as opposed to its behavior, it is best represented in the design model itself as encoded in SLIM. Otherwise, a property (or properties) may be specified for it.

Although properties can be specified by hand in the model, COMPASS also offers a GUI to assist with this. Individual components can be selected, permitting the user to define properties for them. Properties can be specified for component types and their implementation, as well as instances of components that occur as subcomponents of others. This approach adds some flexibility that allows the hierarchy of the model to be altered without invalidating any previously specified properties.

Based on the flow, the first candidate approach to derive a property or properties is using the CSSP. In Figure 5.4, the GUI is presented that is used for this purpose. At the top, the model element to which the properties should be applied can be selected. This can be the component itself, or any of its constituent elements such as modes or subcomponents. Based on the type of element selected, the possible

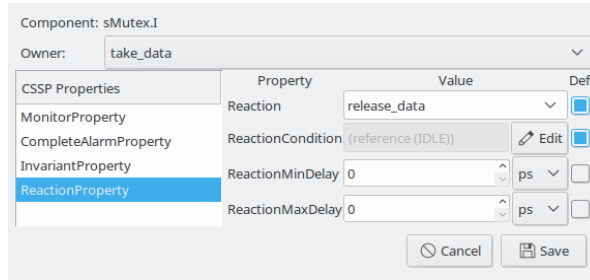


FIGURE 5.4 – Dialog used by COMPASS for CSSP property specification.

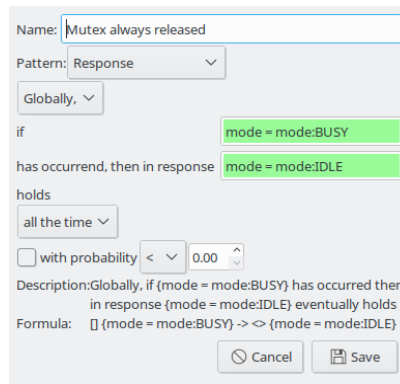


FIGURE 5.5 – Dialog used by COMPASS for pattern-based property specification.

derivable properties are listed on the left. Selecting one allows the associated CSSP properties to be set (or unset). Multiple derivable properties can be configured this way.

If the CSSP can not be used, the pattern-based formalization is the next considered possibility. The GUI available for this purpose is shown in Figure 5.5. In this dialog, the user can make selections and fill in placeholders to construct the pattern in question. Below, both a description phrased in natural language as well as the formal definition are shown. The syntax used for the placeholders is described in section A.6. References to model elements in the placeholders can be made relative to the selected component, and will resolve the specific instances when the instance model is constructed from the root component.

Finally, properties can be specified directly using the language described in section A.6, which supports various temporal operators. Figure 5.6 shows the dialog available for this purpose.

Tying it all together is a wizard which helps select the desired approach based on the flow shown in Figure 5.3. The user is asked to enter a requirement and

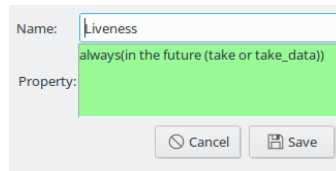


FIGURE 5.6 – Dialog used by COMPASS for manual property specification.

classify it according to the taxonomy presented in Section 3.2. The wizard will then suggest the approach to be taken, and store the requirement and possible properties afterward.

### *Property Types*

When a property is specified, it can be classified as either of the following types:

1. propositional properties;
2. qualitative properties;
3. timed properties;
4. probabilistic properties.

The type of property determines for which type of analysis it is applicable. The primary distinction can be made between properties that describe states and those that describe paths. The prior, which are the *propositional* properties, can be used to express groups of states that share a certain property, which can be used to express desirable or undesirable states, or constraints when exploring the state space.

The other types, which are all a subset of temporal properties, are used for model checking purposes, looking at the behavior of the system. These are the qualitative, timed, and probabilistic properties. The first group considers sequences of states, and uses logics such as LTL and CTL. Timed properties consider in addition the delay between subsequent states, mapping to logics such as MTL and MITL. Finally, probabilistic properties consider the probabilities of transitions between subsequent states, and can be used to measure the overall probability of sets of paths. These map to logics such as PCTL and CSL.

In COMPASS, the logics used are limited to LTL, CTL, MITL and CSL, which are the logics used by the back-end analysis tools. In the following, it will be explained for the various functionalities which types of properties and formalisms are applicable. A more detailed description of these individual logics is given in Appendix A.

### *Contracts*

Contract specifications can be used for the CBA, as will be presented in Section 5.3.4. Like properties, they can be specified in the model manually, but the GUI offers a

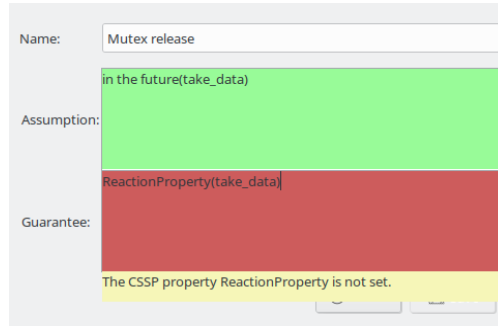


FIGURE 5.7 – Dialog used to specify contracts. The screenshot also shows how errors are indicated, as in this particular case the CSSP ReactionProperty is used but not set.

dialog for this to make this easier (see Figure 5.7). Contracts are specified in terms of an assumption and guarantee, both of which are SLIM propositional expressions as described in section A.6. These may also refer to properties derived from the CSSP.

### 5.3.2 MISSION SPECIFICATION

The mission specification functionality allows various mission characteristics to be modeled and linked to the SLIM specification, such as mission phases, operational modes, and spacecraft configurations.

This function exists currently for traceability purposes mainly, similar to the specification of requirements, and is not linked to any of the analysis capabilities in the toolset. Such a function is left for future work.

The mission characteristics can be specified by first defining the various phases, such as “ground” or “flight”, and operational modes, such as “nominal” and “safe”. Spacecraft configurations are represented by an identifying label and an associated SLIM formula. The spacecraft is considered to be in this configuration if and only if the formula is true. Each configuration can then be association with an (arbitrary) subset of the operational modes defined before.

Finally, observables can be defined for various combinations of phase and operational modes. Each phase can be associated with a subset of operational modes and, for each operational mode, a SLIM formula that evaluates to true if and only if the spacecraft is in that particular configuration of phase and operational mode. Each possible configuration must be uniquely identified by its corresponding formula.

### 5.3.3 CORRECTNESS ANALYSIS

For the validation and verification of the SLIM model itself, the COMPASS toolset offers various functions classified under correctness analysis. These functions allow

inspection of the behavior of the model by means of simulation, inspecting the model for deadlocks or incorrect timed behavior, and finally checking properties.

### *Model Simulation*

A useful feature of COMPASS, to aid in developing and debugging a model, is model simulation. Simulation allows a user to generate traces from the model either randomly or guided by the user. The latter can be performed by explicitly selecting mode transitions that are enabled, or defining the value of variables in the next state, having the model checker resolve everything else. Furthermore, propositional properties can be specified to impose constraints on the states that are reachable.

Internally, this functionality is provided by translating the user input into constraints, either by fixing the next transition or some variable values, as well as conjoining any selected propositional properties. If from the constraints multiple states are reachable, they are selected in a random fashion (hence, random trace generation is simply generating steps without constraints).

### *Model Checking*

Model checking performs an exhaustive check over all possible states in the model, making it possible to conclusively determine whether or not some property of interest holds. The common alternative is testing, which only explores parts of the model based on predefined inputs. As such, testing can only prove the presence of certain behavior, including faults, whereas model checking can prove their absence as well. This makes it possible to prove whether or not the system is safe, by ensuring no unsafe state can be reached.

COMPASS uses the `NUXMV` engine for model checking. To this end, the SLIM model  $M$  is translated to SMV (cf. Section 5.2.3), and LTL or CTL properties are provided in the format understood by `NUXMV`. For each property  $\varphi$ , `NUXMV` will check if  $M \models \varphi$ . If so, an indication will be given to the user that the property holds. If not, a counterexample in the form of a trace will be given.

As described in Section 5.2.4, `NUXMV` can be used for either BDD-, satisfiability (SAT)- or IC3-based model checking. Depending on the method, deadlocks (see Section 5.3.3) may influence the outcome. It is therefore important to ensure the model is deadlock free, or the correct mode of operation is used.

**Fairness** One specific aspect that must be considered for model checking *fairness*. Fairness conditions impose restrictions on the paths that are considered with respect to the events that occur in infinite paths. Generally, such conditions assume that certain events (or their negation) must occur infinitely often on infinite traces. For discrete systems, COMPASS does not impose any fairness constraints. When dealing with timed systems however, COMPASS considers only timed paths in which no two consecutive steps are a time delay, preventing certain kinds of Zeno-

behavior (in particular starvation of discrete events). Such behaviors are discussed in more detail in the following.

### *Deadlocks and Timed Correctness*

An important caveat of models is that due to their abstract nature, it is possible that they describe unrealistic behavior. This is often unexpected and hard to detect manually. In addition, they may result in misleading or even incorrect results from the tools in COMPASS.

One of the most well-known cases is the *deadlock*. In a deadlock, the model is in a state from which no other state is reachable. Depending on the formalisms being used, this may be acceptable or not. Within COMPASS however, a deadlock is a modeling error. This holds true in particular for *reactive* systems, where the system should always be receptive to new input. COMPASS offers the ability to check for deadlocks in models with a finite state space. By constructing the finite-state model (FSM), `NUXMV` can check for states which have no outgoing transitions.

Ensuring the model is deadlock-free is particularly important in case BDD-based analysis is performed, as in this case only paths of infinite length are considered. It can thus occur that a path leading to the invalidation of some property is ignored if it leads to a deadlock state.

More complicated is the occurrence of invalid timed behavior in hybrid models. This entails *Zeno behavior* and *time-/clock-divergence*. Zeno behavior refers to traces in which an unbounded number of discrete steps occur within a finite time delay (i.e. the sum of delays of timed steps is finite). This can occur if there is a loop in the system which does not enforce time to progress when iterating. A particular case of such behavior is the *time-lock*, which is a state in the system from which no path can be extended that permits time to increase (similar to a deadlock, but discrete state changes are still possible).

Zeno analysis is performed based on three user inputs: the mode to check, a step bound, and a time bound. The analysis is performed by checking if the mode is reachable, and whether Zeno behavior can be detected starting from this mode. The latter is done by checking for paths of infinite length starting from this mode, based on the given step and time bounds<sup>2</sup>. If there is no infinite path starting from the mode that exhibits Zeno behavior, the mode is classified as non-Zeno. The analysis can provide the following results for individual modes:

- » Zeno: A Zeno path can be found starting from the mode. In this case a Zeno trace is given that reaches this mode.
- » Non-Zeno: No Zeno path can be found. In this case a non-Zeno trace of infinite length that reaches the mode is given.
- » Unreachable: The mode cannot be reached. In this case no trace is given.

---

<sup>2</sup>Without these bounds, checking for Zenoness is undecidable.

- » Unknown: If either the step- or time bound is exceeded, the result is unknown. The analysis should be performed again with higher bounds.

The other behavior to be avoided, time-divergence, is also treated by the COMPASS toolset. Time divergence occurs if the value of a clock is allowed to grow unbounded. This can be problematic, as SAT-based approaches require loops in order to reason over paths of infinite length. If a clock value is never reset, such a loop does not exist. Time divergence analysis operates by searching for paths in which the value of some clock exceeds a predetermined threshold. The user is asked to provide a clock (or clocks) and a bound to compare it against, as well as a step bound. The result of time-divergence analysis can be any of the following:

- » Unbounded: The clock can reach a value that exceeds the given threshold. In this case, a trace is provided that leads to a state where this is the case.
- » Bounded: The clock's value does not exceed the threshold.
- » Unknown: No path can be found up to the step bound where the clock's value exceeds the threshold. However, it is not proven that no path exists where this is not the case. In this case the step bound can be increased to try and get a conclusive result.

#### 5.3.4 CONTRACT BASED ANALYSIS

The analysis so far has considered the model as a monolithic whole. Alternatively, a *component-based* approach can be taken, where the individual parts of the system are being considered. This approach allows for the application of compositional reasoning, incremental refinement as well as the reuse of components [18].

With compositional reasoning, the analysis of a system is reduced to that of its constituent components. This makes it possible to analyze larger models. One of the major risks of model checking larger systems in the *state-space explosion*, in particular when the composition of various systems is considered. Being able to run analysis on the various (sub)systems individually permits the size of the model under consideration to be kept manageable.

Reuse of components makes it also possible to reuse any proof associated with them. Thus, an externally provided component with proven guarantees may be used in the system with little extra cost in terms of formal analysis. Being able to refine the system in a step-wise manner also fits well within the scope of aerospace projects, as well as others. This has been discussed in Section 3.3.3.

Within COMPASS, an assume-guarantee based approach[83] is taken for compositional verification. The assumptions and guarantees of components are specified pairwise in *contracts*, defined at the interface of a component.

COMPASS supports three types of analysis for the contracts that may be specified on the model, which will be explained in further detail in the following sections:

- » Validation, where *validation properties* are checked against the specification;

- » Refinement Checking, where the contract refinements are verified;
- » Tightening, where for individual contracts it is checked if they can be *tightened*, which is the relaxation of assumptions and/or the strengthening of guarantees.

### *Validation*

Contract validation makes use of *validation properties*. These validation properties are specified on contract refinements, and are defined in terms of assumptions, guarantees and norm guarantees of contracts. Here, norm guarantees are defined as  $A \rightarrow G$ , where  $A$  and  $G$  are the assumption and guarantee of a contract respectively.

Three types of validation property can be specified, which are consistency properties, possibility properties, and entailment/assertion properties.

Consistency properties are used to check for the absence of contradictions between validation properties in a given subset. If the validation properties are mutually satisfiable, a *witness trace* is generated that shows its satisfiability. Otherwise, it is possible to generate an *unsatisfiability core* (or simply “unsat core”) which provides the corresponding counterexample.

The possibility properties check for consistency of a user-specified property with respect to the (subset of) validation properties. This can be used to check that the validation properties permit a specific kind of behavior. The result is similar to that of a consistency check, being either a witness trace or unsat core.

Finally, the entailment (i.e., assertion) properties check if a user-specified property is implied by the (subset of) validation properties. This permits the user to check if the properties defined by the contract(s) actually entail some expected property of the system. If this is not the case, the properties of the contracts may have to be extended, or a new contract may be added.

Note that these checks are aimed primarily to validate the contracts themselves, as opposed to the system they are specified for. Similar to requirements validation, they help ensure the specification is written down correctly.

### *Refinement Checking*

The contract refinement checking function verifies that the contract refinements specified in the model hold, that is

- » if the contracts of the subcomponents are satisfied, then so are those of the composite component, and
- » if the contracts of the subcomponents are satisfied and the environment of the composite component satisfies its assumptions, then the assumptions of each subcomponents are satisfied as well.

The OCRA tool will generate the required proof obligations for this purpose, and reports that either the requirement is satisfied (optionally bounded), or provides a counterexample.

### *Tightening*

The contract tightening functionality allows for automatic weakening or strengthening of assumptions and guarantees. The main benefit is that this releases some of the burden from the designer of the specification, as this permits the specification of the contracts to be simplified.

Contract tightening is implemented by means of parameter synthesis as described in [43]. In summary, parameters are injected into the contract assumptions and guarantees, which are translated into the, now parameterized, proof obligations. The result can be treated as a parameter synthesis problem, where parameters are chosen to find the weakest conditions such that the contract refinements are still satisfied.

Two tightening approaches are available, *top-down* and *bottom-up*:

- » Top-down tightening works by weakening the assumptions of composite components and the guarantees of child components;
- » Bottom-up tightening works by strengthening the assumptions of child components, and the guarantees of parent components.

Executing either of these functions will result in the (possibly) tightened contracts, which can subsequently be applied to the specification by the user.

### 5.3.5 PERFORMABILITY

The *performability* [105] functions of COMPASS offer performance and reliability analysis by means of probabilistic model checking. By transforming the SLIM model into a CTMC or IMC (depending on the presence of non-determinism, see below), numerical analysis is possible using either MRMC or IMCA. Alternatively, the SLIM specification can be subjected to statistical analysis using the Monte-Carlo method, the functionality of which is provided by SLIMSIM.

In either case, a reachability property is formulated for which the probability of it holding true is determined. In the case of IMCA, it is also possible to determine the expected time and long-run average of reaching or remaining in the target state(s). The following sections explain these processes in more detail.

### *Numerical*

The numerical approach to performability analysis uses the MRMC and IMCA model checkers to perform the probabilistic analysis of the CTMC or IMC underlying the model. In order to derive these, first the state space of the model has to be generated.

The state space is generated by `nuXmv`, the successor of `NuSMV`. It is given the SLIM model as input, and calculates the *symbolic* state space in terms of a BDD.

The pivot between the model as handled by `nuXmv` and the IMCA and MRMC model checkers is `MODEL2IMC`. `MODEL2IMC` takes a BDD representation of the model, and applies the necessary transformations to make it suitable for MRMC or IMCA as described in Section 5.2.3. `MODEL2IMC` can also handle DFTs as input, which will be described in Section 5.3.6.

At this point it is technically possible to derive an IMC directly, however, the state space at this point is generally prohibitively large. In order to reduce it, *symbolic bisimulation* is applied using the `SIGREF` framework [141]. `SIGREF` is applied to the BDD (annotated with the error rates) in conjunction with the properties of interest (see Section 5.2.2) in order to minimize it.

If the result is fully deterministic, a CTMC is constructed, otherwise an IMC. CTMC models can be exported to the file formats supported by both MRMC and IMCA. The IMC models are supported by IMCA only. As MRMC provides better efficiency for the analysis of CTMC models, COMPASS will prefer it automatically if possible.

When using MRMC, the property is provided using the CSL syntax of MRMC. For IMCA, the property is encoded by specifying the goal states in the IMC input file and the time bounds using program parameters. As IMCA can handle non-deterministic models, both the minimum and maximum reachability probabilities are calculated.

For these reachability properties, it is also possible to provide the full cumulative distribution function (CDF) up to the provided time bound when the start time is zero. This gives an idea of how the reachability probability changes over time. An example plot is given in Figure 5.8.

In addition to reachability queries, it is also possible to calculate the *expected time* or *long-run average* for state-based properties [70]. This functionality is provided by the IMCA tool.

The expected time is determined by the mean sojourn time of the states in the model. Starting from the initial state, for each path leading to the goal state(s), the sum of these sojourn times is taken, weighed according to the probability of taking that path.

For the long-run average, first the maximal end components (MECs) are generated. Of these MECs, the ratio between the sojourn time of the goal states and non-goal states is determined. Then, from the initial state the probability to reach each MEC is determined, which is used to determine the overall weighed average fraction of time spend in goal states.

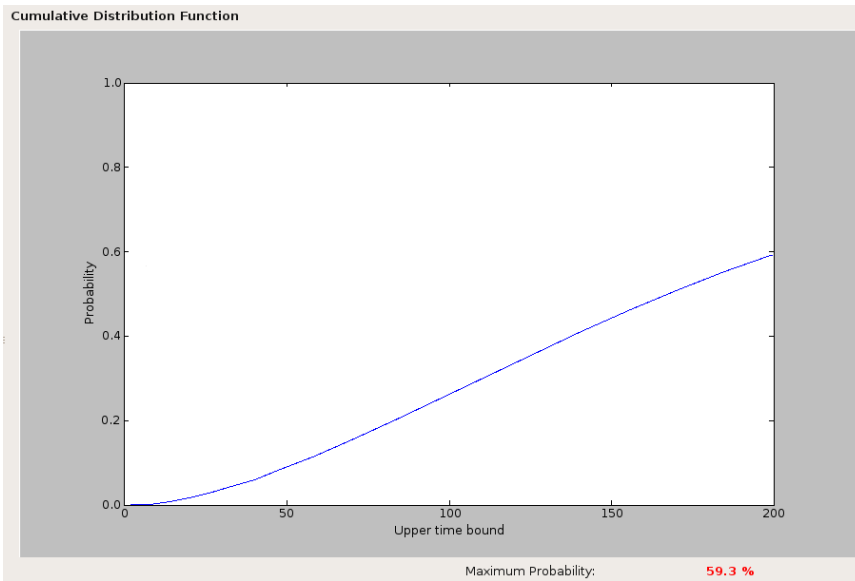


FIGURE 5.8 – Example CDF as provided by performability analysis.

### *Simulation*

New in COMPASS 3.0 is the possibility to run performability analysis based on Monte-Carlo simulation. This process depends on the functionality described in Chapter 4, provided by the SLIMSIM tool.

As SMC does not require the entire state space to be constructed, neither SIGREF nor NUXMV/NUSMV are used. Rather, the model is directly translated into a format suitable for SLIMSIM.

Whereas for the use of MRMC or IMCA only a single parameter, the error bound, is given, SMC requires more: Aside from the error bound, also a confidence parameter and strategy are required. The function of these is explained in Chapter 4.

As opposed to the approaches using Markov chains, the simulator only determines the probability at the upper time bound. Hence, a CDF is not plotted, but only the final probability estimate is given.

### 5.3.6 SAFETY AND DEPENDABILITY

The safety and dependability functions of COMPASS provide insights into possible failure modes of the system, their interaction and how well the systems responds to them.

Both fault tree analysis (FTA) and FMEA rely on the generation of minimal cut-sets (MCSs). A *cut set* is a collection of error events the combination of which will

trigger some other (top-level) event of interest, such as a failure. A *minimal cut set* is one of the smallest possible sets of such events, such that the removal of any events from such a set will cause the event of interest to no longer occur. The size of a cut set is referred to as its *cardinality*.

### *Fault Tree Analysis*

Fault trees are a well-known concept in safety engineering, and provide a graphical overview of the possible interactions and dependencies between various faults in the system, eventually leading to a specific failure mode. Fault trees, as the name implies, are classically tree structured graphs, though this can be extended to directed acyclic graphs (DAGs). At the bottom the nodes represent *basic events*, which are the base faults that may occur in the system. At the top is the node which represents the *top-level event (TLE)*, which is a specific failure mode. The nodes in between represent various (logic) gates, which are triggered by specific combinations— and possibly orderings— of child nodes, thus describing how faults may propagate.

COMPASS provides the ability to automatically generate fault trees from the SLIM specification, perform quantitative analysis on them to calculate failure probabilities, and provide measures of fault tolerance.

**Generation** Fault tree generation requires as input the (fault-extended) SLIM model and a failure condition as TLE  $\varphi$ . The error events and propagations in the SLIM model are considered as the possible error events leading to the TLE, and are represented as basic events in the fault tree. The root node of the fault tree represents the states which satisfy the TLE  $\varphi$ . The combinations of events that lead to the TLE are determined based on the possible MCSs that lead to  $\varphi$ . To this end the xSAP [25] tool is used.

xSAP determines the possible MCSs by a forward or backward search, as given in [32]. Starting from either the initial state (forward search) or the TLE (backward search), the reachable state space is computed by expanding the frontier incrementally. For each failure mode, a history variable keeps track of its occurrence. When the complete reachable state space has been explored, the cut sets are determined by projecting the intersection of the reachable state and the TLE over the history variables. These cut sets are minimized to determine the final set of MCSs. The elements of the MCSs correspond to the error events in the model, and with that the basic event nodes in the fault tree.

Each MCS is represented using a new AND-gate in the tree, the children of which are the basic events corresponding to the events contained in the MCS. The collection of AND-gates is then given as the children of the OR-gate that represents the TLE. For cut sets of cardinality one, the basic event is connected to the TLE node directly.

An extension of this procedure is the generation of DFTs, which takes the ordering of events into account. The cut sets resulting from analysis using xSAP are ordered,

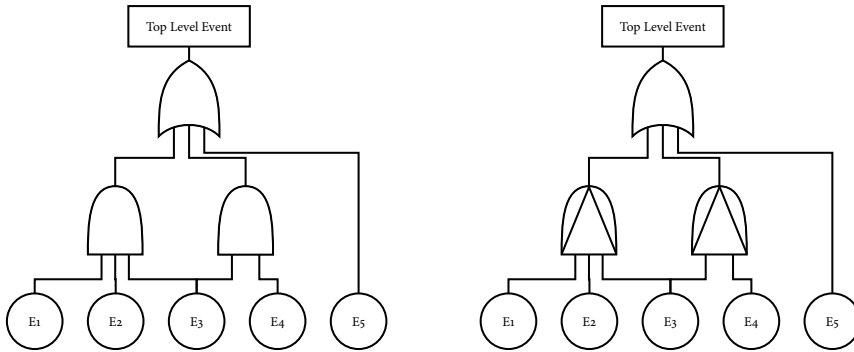


FIGURE 5.9 – Two example fault trees as can be generated by COMPASS. On the left is a static fault tree, on the right a dynamic fault tree. Here, the TLE, triggered by an OR-gate, fails if either of the two (P)AND-gates fail, or basic event  $E_5$ . The (P)AND-gates have 3 and 2 children respectively, of which one is shared.

yielding cut *sequences*, and are mapped to PAND-gates in the tree.

Figure 5.9 depicts some example fault trees as they could be generated by COMPASS. On the left is a static fault tree that does not take ordering of events into account. On the right is a dynamic fault tree that does. The TLE is triggered by an OR-gate, which itself is triggered by either of two (P)AND gates, or basic event  $E_5$ . The (P)AND gates are triggered by the basic events  $E_1, E_2$  and  $E_3$ , or  $E_4$  and  $E_5$  respectively. For the PAND-gates, the ordering of the occurrence of the basic events matters, requiring them to occur, in the diagram, from left to right. A complete description of static and dynamic fault trees can be found in sections D.2 and D.2.1 respectively.

**Hierarchical generation** The approach to generate fault trees so far generates fault trees based solely on the set of error events in the model, ignoring its structure. A negative side effect of this is that the model’s structure is not reflected in the fault tree, resulting in a flat structure with OR-gates on one level, and AND-gates below. Using CBA a hierarchical approach is possible [34], which results in fault trees of which the structure follows the hierarchy of the model much more closely, making them more similar to fault trees designed by hand.

First, faults are injected into the components and contracts, representing the cause and propagation of failures. This step is performed after the model has been translated to OSS. For each component  $S$ , an input and output port are added representing the failure, which are  $f_S^I$  and  $f_S^O$  respectively. The contracts are then extended to include these failure conditions to weaken them. A contract pair  $(A_S, G_S)$  is replaced by  $(\neg f_S^I \rightarrow A_S, \neg f_S^O \rightarrow G_S)$ . Next, the port connections are extended. The output failure port depends on the MCSs driving the failure of  $S$ ’s guarantee based on the guarantees of its subcomponents and environmental input. The input failure

port  $f_U^I$  of each subcomponent  $U$  depends in the MCS driving the failure of the assumption  $A_U$  based on the failure of other subcomponents or the environment.

The fault tree is generated using a recursive approach, starting from the TLE. The TLE is associated with the output failure port of the root component. If the failure does not depend on that of subcomponents, it is treated as a basic event. Otherwise, children are recursively added depending on the MCS that was determined for the port connection.

For each cut set of a component, an AND-gate is created for each of its children. Between cut sets, and the basic event corresponding to the failure of the environment of the component, an OR-gate is created. A simplification is applied such that nodes with a single child are removed and the child is added to its parent.

**Evaluation** Fault tree evaluation determines the probability of the gates being triggered over a given time-span. In order to do so, each basic event in the fault tree has to be associated with an error rate, which is derived from the SLIM model. Using the translation as described in Section 5.2.3, a probabilistic model is derived from the fault tree.

Given a time bound  $d$ , for each gate  $g$  a reachability property is derived of the shape  $\psi_g = \top \mathcal{U}^{[0,d]} g$ . Using MRMC, or IMCA in case the fault tree is non-deterministic, the reachability probability is determined and associated with the corresponding gate.

**Verification** Fault tree verification is a generalization of fault tree evaluation. Instead of determining the reachability probability of the TLE, the user is permitted to provide any probabilistic property referring to the gates of the fault tree in the atomic propositions. This makes it possible to provide arbitrary Boolean combinations of gates in the atomic propositions of the properties.

Instead of associating the resulting probability with the generated fault tree, as is done for evaluation, the resulting CDF is plotted similar to how it is done for performability analysis (cf. Section 5.3.5). This not only provides the overall reachability probability, but also its progression over time.

#### *Failure Mode and Effect Analysis*

A well-known technique in the area of safety assessment is FMEA, where for some failure condition the possible causes are determined. Based on the same MCS generation as used for fault trees, COMPASS can generate FMEA tables, where each MCS attributes to an entry in the table.

The cardinality of the MCS to be considered is set prior to generating the table. This means that an FMEA table is generated for a specific cardinality, but allows the search for MCSs to be simplified, as lower cardinalities have to consider less faults.

Similar as with fault trees, the ordering of events can be taken into account, which in COMPASS is referred to as a *dynamic* FMEA table. Furthermore, the table can be *compacted* by detecting cut sets which are a proper subset of other cut sets. These larger subsets can then be left out, since its failure mode is already implied by the subset.

An extension of FMEA is failure mode, effects and criticality analysis (FMECA), which considers the criticality of the possible failure effects. However, this is not considered in COMPASS.

### *Fault Tolerance Evaluation*

Fault tolerance evaluation determines the number of unique cut sets in a set of fault trees that have been generated prior. In other words, for a set of TLEs the possible *unique* cut sets are determined, grouped according to their cardinality, giving an overall indication of the fault tolerance of the system.

#### 5.3.7 FAULT DETECTION, ISOLATION AND RECOVERY

An important aspect of RAMS analysis is FDIR. COMPASS offers various features to support FDIR analysis, backed by the xSAP platform.

FDIR analysis relies on the definition of a *failure condition* and one or more *alarms* in the model. The failure condition is a propositional property describing the states in which the system is considered failed. An alarm is a Boolean data port that changes state as soon as some fault is detected in the system. Such ports are marked with the `Alarm` property. The COMPASS FDIR analysis functionality aims to prove which faults cause which alarms to trigger.

The two primary means to do so are fault *detection* and fault *isolation* analysis. The prior indicates which alarms are triggered by the occurrence of some failure condition. The latter results in a fault tree that shows which combinations of faults in the model (i.e., error states) can lead to some alarm being triggered. Finally, fault *recovery* analysis checks whether or not some failure condition can be recovered from.

### *Fault Detection*

Fault detection analysis can be used to determine which alarms will be raised depending on the failure condition property. To determine which alarms are raised, a model-checking problem is phrased that checks for each alarm if its state differs from the initial state when the property to be diagnosed no longer holds, i.e.,  $\Box(\varphi \rightarrow (\varphi \mathcal{U} o \neq o_I))$ , where  $\varphi$  is the property to be diagnosed,  $o$  is the alarm's current state, and  $o_I$  the state of the alarm in the initial state.

Optionally a delay bound can be given, which defines the maximum delay that is permitted to consider an alarm as a means of detection for the failure condition,

i.e., the range  $I = [0, d]$  with  $d$  the detection delay. This range is applied to the until-operator in the above model checking problem, yielding a formula in MITL.

### *Fault Isolation*

Fault isolation determines which errors lead to some alarm being triggered. For each alarm a fault tree is constructed, with the SLIM error events that allow the alarm to be triggered as the basic events, and the alarm itself as the TLE, using the same condition  $o \neq o_I$  as from fault detection.

### *Fault Recovery*

Fault recovery analysis determines whether some recovery property holds. This is a simple model checking problem that can be expressed by a check that some recovery state, as described by the recovery property, can be reached after a fault state has been visited. For this purpose the *response* property patterns are particularly useful (cf. Section 3.3.1).

## 5.3.8 DIAGNOSABILITY ANALYSIS

Diagnosability analysis concerns itself with the question whether or not a system is able to correctly diagnosing a fault. A fixed amount of observable data is available, generally a small subset of all possible data depending on the availability of sensors, which the system has to use to derive the possible cause for a fault. If insufficient observable data is available, diagnosing a fault may not be possible. On the other hand, the amount of observable data is minimized in order to reduce cost and complexity.

Diagnosability analysis checks if a set of *observables* is sufficient to detect a failure condition. More specifically, a failure is considered diagnosable if the state of the observables always differs between states. If there exist two paths in the system that lead to the same observations, but with one having the failure condition triggered and the other not, the failure is considered not-diagnosable. These paths can then be provided as a counterexample [45].

Diagnosability in COMPASS relies on the definition of *observables* in the SLIM model, which are data ports in the model from which a diagnoser can consider the value when determining what the current fault condition is. An observable is defined in a SLIM model by tagging a data port with the *observable* property. This is combined with a failure condition property similar to those specified for FDIR analysis (in fact the same properties can be used).

The analysis uses the *twin-plant approach* [45]. With this approach, two versions of the model are composed together, one being the original, the second with all state variables relabeled to primed variants. These are designated  $M$  and  $M'$  respectively. These models then contain the observable sets  $\{o_1, \dots, o_n\}$  and  $\{o'_1, \dots, o'_n\}$  respectively. The property  $\varphi$  for which diagnosability is to be proven is also duplicated

with renamed variables, becoming  $\varphi'$ . Diagnosability then becomes a model-checking problem checking the reachability property  $M||M' \models \diamond(o_1 = o'_1 \wedge \dots \wedge o_n = o'_n \wedge \varphi \wedge \neg\varphi')$ . If the property holds true, it means a state exists where the observables have the same value, but the failure condition can either hold or not, meaning it is not diagnosable.

As the computation on the parallel composition is potentially computationally expensive, the problem can be simplified by the addition of a diagnosability context, which is a property describing the states under which diagnosability should be considered, e.g., a safe mode. For such a property  $\psi$ , the model-checking problem then becomes  $M||M' \models \diamond(\psi \wedge \psi' \rightarrow o_1 = o'_1 \wedge \dots \wedge o_n = o'_n \wedge \varphi \wedge \neg\varphi')$ . It is also possible to add *path restrictions*, which are a set of events that have to occur in a specific order for a path to be considered for diagnosability analysis, further reducing the size of the state space to be searched.

It is possible to provide a diagnosability delay  $d$ , which defines the maximum delay between the occurrence of the failure condition and its diagnosis. This delay is explicitly encoded in the model as a timer  $c_d$  that starts as soon as the failure condition is triggered, as well as an invariant property  $\neg(c_d \geq d)$ . This invariant corresponds to states where the delay bound has been exceeded since triggering the failure condition, but the observations have not yet diverged.

### 5.3.9 TIMED FAILURE PROPAGATION GRAPHS

The support for TFPGs [23] adds the ability to analyze the temporal dependencies between failure modes and their effects on a system. Structurally there are some parallels with fault trees. However, whereas fault trees are concerned with the likelihood of possible failure propagations, TFPGs consider the time delay between fault propagations. A full description of TFPGs can be found in section D.3.

COMPASS provides three main functions that make use of TFPGs, namely synthesis, behavioral validation, and effectiveness validation. Synthesis concerns itself with deriving a TFGP from the SLIM model. Behavioral validation checks that a TFGP matches with a SLIM specification in terms of accepted behavior. Effectiveness validation check that the various failure modes of a TFGP can be diagnosed in the model.

**Linking to SLIM** To perform any kind of TFGP analysis based on a SLIM model, first some associations with the SLIM model have to be made. These associations link information from the SLIM model to TFGP failure modes and discrepancies (although at that point a concrete TFGP does not yet have to exist).

First, SLIM error states can be linked to TFGP failure modes. This link associates entering such an error state in the model with the triggering of the corresponding failure mode.

Next, SLIM expressions are linked to discrepancies. These are Boolean expressions that trigger the discrepancy when they become true (i.e., they are true in the current

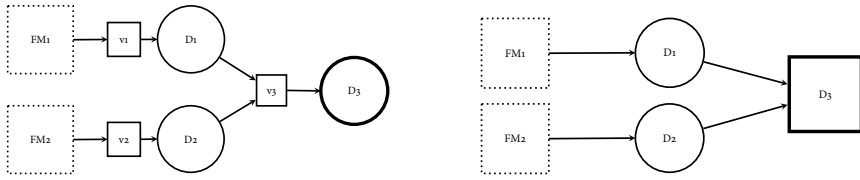


FIGURE 5.10 – Two generated TFPGs. On the left the TFPG has not yet been simplified, on the right it has.

state, whereas they were false in the previous). These links are subdivided between monitored and unmonitored discrepancies.

Finally, associations are made between TFPG modes and SLIM expressions. When an expression evaluates to true, the mode is considered active, and any propagation edge that is associated with that mode becomes enabled.

Note that for discrepancies and TFPG modes, the association SLIM expressions do not have to refer to SLIM modes, although this is usually the case.

### Synthesis

Given a set of TFPG associations with a SLIM model, it is possible to synthesize a TFPG. At the basic level a similar procedure is used as for the synthesis of fault tree, using FTA. The complete algorithm is described in [23], an overview is given below.

For each discrepancy defined in the SLIM associations, an OR discrepancy node is added to the graph. Similarly, for each failure mode association, a failure mode is added as well. Then, using FTA, the MCSs are determined for each discrepancy node. Each cut set  $mcs$  is mapped to a new virtual AND discrepancy node  $v$ , with an edge leading from the discrepancy or failure mode in  $mcs$  to  $v$ , and an unconstrained edge from  $v$  to the discrepancy node for which  $mcs$  was determined.

**Simplification** The generated TFPG is both complete and correct. It can, however, be simplified. First, some edges towards the virtual AND discrepancy nodes can be removed. Edges can be removed if they do not alter the triggering conditions of discrepancy nodes. After the removal of these edges, redundant AND nodes may be removed as well. AND nodes with an in- and out-degree of one can be removed, replacing it with an edge between the source and target nodes. If two AND nodes share the same inputs and outputs, one of them can be removed from the graph, as well as its connecting edges. Finally, an OR node with only one source edge coming from an AND node can be simplified by merging it with this AND, the resulting node being of AND type.

Figure 5.10 shows an example of a generated TFPG before and after simplification. The AND nodes labeled  $v$  are the virtual AND nodes, the other nodes are respec-

tively the failure modes (dotted, square) and OR discrepancy nodes (solid, round). The simplification permits the  $v1$  and  $v2$  nodes to be removed, and the  $D3$  and  $v3$  nodes to be merged.

After simplification, time bounds and mode conditions can be tightened. Tightening of time bounds is performed based on parameter synthesis and IC3. Time bounds on edges are incrementally brought closer together until they are sufficiently tight. For details, refer to [24].

### *Behavioral Validation*

With TFPG behavioral validation, a check is performed to ensure the TFPG captures all the possible behaviors from the SLIM model with respect to the associations that have been made. This is done by ensuring all the traces in the model have a corresponding trace in the model of the TFPG.

The associations provided by the user are used to map the traces generated by the SLIM model to the traces of the TFPG model. Behavioral validation then checks if the traces that are generated by the SLIM model have a corresponding trace in the TFPG. To this end, proof obligations are generated that are given to the xSAP tool for verifications. These proof obligations are constructed such that they ensure an OR discrepancy node is activated only after at least one incoming discrepancy is triggered, after at least a delay of  $t_{\min}$  time units, where  $t_{\min}$  is the minimum delay associated with that incoming edge. Another proof obligation is defined to ensure the propagation delay is no more than  $t_{\max}$  time units, where  $t_{\max}$  is the maximum delay associated with an incoming edge. For AND discrepancies, similar obligations are generated, but then considering the conjunction of all conditions on edges, rather than their disjunction.

Note that the behavior of the TFPG can be considered an over-approximation of the SLIM model. That is, any behavior occurring in the SLIM model can be proven to be captured by the TFPG using behavioral validation, but the TFPG may also model traces that do not occur in the SLIM model.

### *Effectiveness Validation*

Effectiveness validation can be used to check if failure modes of a TFPG are diagnosable using observable discrepancies, similar to the diagnosability of faults in the SLIM model as presented in Section 5.3.7.

For each combination of failure mode and system mode in the TFPG, diagnosability analysis is performed. The monitored discrepancies, system mode and overall time progression are considered the observables, with the status of the failure mode the condition to be diagnosed. As for diagnosability of SLIM model, the twin-plant approach is used, with as input an SMV model translated from the TFPG.

For each combination of failure mode and system mode, the outcome can be either:

- » not diagnosable, in which case a critical pair of traces is given with the same state of observables, one without the failure mode triggered and one with;
- » no counterexample found, in case SAT-based model checking is used, and no counter-example was found within the SAT-bound;
- » diagnosable otherwise.

## 5.4 FUTURE WORK/ROADMAP

Although COMPASS already provides a fairly extensive set of features, there are still possibilities for improvement. As part of the COMPASS 3 project, a Roadmap was defined in which these points were identified [36]. Various objectives have been identified pertaining to the toolset itself, its associated development process, various research directions, outreach to the community, and further integration with other ESA initiatives. This section will give a brief overview of these.

First, various objectives are given to improve the toolset itself. Feedback from industrial partners mostly indicated a wish for better interoperability with other design languages and tools. This led to the definition of the following objectives:

- » Better alignment of the SLIM language with AADL and its annexes. In Chapter 2, various differences between SLIM and AADL have been identified, and better interoperability can be achieved by removing or minimizing them.
- » Provide the ability to use different formalisms and languages for modeling system specifications, meaning COMPASS supports more languages than just SLIM, such as Simulink [126] or SysML [132]. The project intended to do this is currently dubbed COMPASS\*.
- » Better integration possibilities for other design environments, such as OS-ATE [113] (or other Eclipse-based environments), MATLAB or AutoFOCUS3 [11].
- » Automated validation of models, and generation of documentation. Examples of such functions are the automatic comparison of (formal) models, change management support such as providing traceability information, and generating artifacts such as a visualization of the input model using state machines or message sequence charts.
- » Improvements in scalability, which requires more efficient analysis engines, for which further research is required.

On the topic of research, various extensions of existing approaches, as well as new functions based on ongoing research are proposed:

- » Improved property validation, to support checking the consistency not only of linear time properties, but also branching (including probabilistic) properties. Other validation techniques can analyze the realizability of properties,

- checking if there exists an implementation that can fulfill it. Finally, another interesting direction is providing the capability of synthesizing tighter bounds on delays or probabilities, which aids in understanding and optimizing the properties.
- » More precise contract-based fault tree generation. Currently, the contract-based fault tree generation is fairly pessimistic and ignores the fault injections specified in the SLIM model. Integrating the two approaches would solve this.
  - » Extended dynamic fault tree analysis. Currently, the dynamic fault trees generated by COMPASS use only Priority-AND gates. However, other gates may give a better intuition of the possible failure modes. Furthermore, advancements in the state-of-the-art of state-space analysis of fault trees allow for larger models to be analyzed more efficiently, which could be implemented in COMPASS as well.
  - » Various improvements and opportunities have been identified for analysis of the FDIR design. Recent work on diagnosis and FDIR [33, 35] can be incorporated into the toolset, for instance by adding the ability to take diagnosability delays in more contexts into account. Further research is being performed to support the automatic generation of observables sufficient for diagnosability. The automatic FDIR component synthesis, as was introduced in the FAME project, can also be incorporated into COMPASS. Finally, TFPG analysis can be further extended to support the production of TFPGs with tighter bounds, generated automatically from the specification.
  - » The analysis of alternative designs — design space exploration — is particularly relevant during early design stages, where the effect of reliability of different designs is compared. Some work has been done to do so in a formal setting [62, 100], and integrating this into COMPASS would be beneficial.
  - » Introducing parameters into the model allows for algorithms to be used that can automatically *synthesize* values for these parameters that fulfill some property, making it possible to choose them in an optimized fashion [HB5]. Furthermore, they permit the application of model *repair*, where values are automatically adjusted such that the model can be made to fulfill some property.
  - » Currently, probabilistic analysis concerns itself with only a single metric for analysis, usually the reachability property for goal states. With multi-objective verification [119], it is possible to consider multiple metrics at the same time, such as a cost to reach the goal and the time to reach it. This makes it possible to look for an optimal weighting of these metrics.
  - » Currently, COMPASS offers no functionality to verify an actual implementation against the model used for verification. With model-based testing it is possible to do so [65, 131], comparing an actual hardware or software implementation against the expected behavior from the model.

## 5.5 DISCUSSION

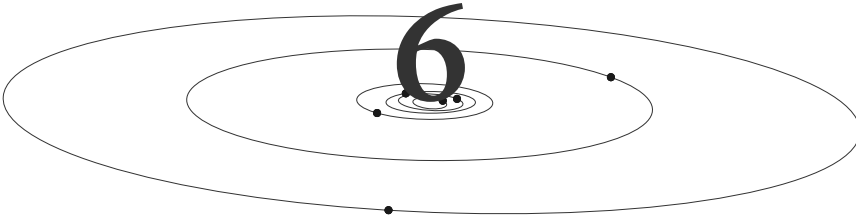
For safety-critical systems, verification is one of the most important aspects of its design process. Yet, with today's practices a lot of work in this area is done by hand. Similarly, the documentation associated with it is also manually provided, including traceability aspects. Finally, since no automatic process is in place, all these artifacts will have to be validated and verified manually, which can be a significant burden, or partially skipped due to resource constraints.

This leads to various problems. Most importantly, there is the risk of not covering all possible cases due to sheer complexity or by mistake. Mismatches between design and implementation, and between documentation and design may also occur. Finally, an update of the design (or requirements) requires all documentation and models to be updated as well, which can easily lead to oversights. Automation of these tasks mitigates these problems and reduces the overall workload. However, care should be taken that this is not done by various loosely coupled approaches, as this yields redundant work to provide the invariably different inputs from the same design.

The COMPASS toolset was created to address these issues specifically, as well as provide a platform to investigate future improvements in this area. This chapter gave an overview of its capabilities, and a roadmap that provides an overview of current and future developments that are applicable. COMPASS is not the only endeavor in this field. Other similar projects include TASTE [133], or those that define standardized architectures such as CORDET [52] and its follow-up projects. This shows there is a significant drive to improve the status quo.

Of course, limitations remain. One of the main problems is that of scalability, as large models quickly require significant computational efforts, or require semantics for which analysis algorithms end up being undecidable. Practical applications can choose to find a subset of desirable functionality that remain within reach, trade formal rigor for efficiency, but foremost desire new scientific results that bring the desired functions within reach.

In another direction, the technology must be adopted by the industry for it to be useful. Existing practices require COMPASS to bring significant advantages to overcome the cost of implementing this change, or lower the effort required to adopt. New features and increasing the technology readiness level can make the use of COMPASS seem more advantageous. Bringing closer integration with existing methods can lower the effort required to adopt. Previous workshops held by the ESA with industrial partners identified some approaches to make such improvements effectively, as discussed in the roadmap.



## COMPASS DEMONSTRATOR

*ABSTRACT – This chapter describes the construction of a demonstrator for the use of the COMPASS toolset in order to make its purpose more tangible. It uses a model-driven approach to generate code for a LEGO® model of a satellite. A translator is used to convert a SLIM instance model into a Simulink model. This Simulink model can subsequently be used to generate executable code for the LEGO® Mindstorms platform, which is used to control the LEGO® satellite model.*



**S**TARTING from a SLIM model, the functions of COMPASS discussed so far only permit the analysis of this model. However, using it to derive an implementation is certainly feasible. Making use of this fact, an approach was developed that permits a SLIM model to be deployed on a physical device, specifically a robot made using LEGO® Mindstorms, allowing the abilities of COMPASS to be demonstrated in a tangible way. To do so, an MDE workflow was engineered and implemented, which is described in full detail in [9].

The flow starts with translating SLIM to Simulink, which is a modeling language available within MATLAB. Simulink is well known in the industry, not in the least for its support to automatically generate (C) code from the model. A particularly interesting feature is its ability to generate C code that is suitable for the LEGO® Mindstorms platform, which can be used to interact with various sensors and actuators.

To demonstrate the approach, a model of a satellite was built with LEGO®, featuring the controller and various peripherals from the Mindstorms platform to provide interaction with the environment. Subsequently, a SLIM model was created to model the logic that should control the satellite, which with the help of Simulink could be deployed on the Mindstorms platform, and executed. This shows that a single

input model, specifically one written in SLIM, can be used both for formal analysis using the COMPASS toolset (cf. Chapter 5) as well as a basis for implementation.

## 6.1 SIMULINK

In order to derive executable code from SLIM, Simulink was chosen to be an intermediate modeling language. It is well known in industry and has a proven track record of being able to export to executable code, in particular to the Mindstorms platform, which is very accessible in terms of prototyping and displaying the model. This makes it an interesting target for translation.

Simulink is a graphical language, where the basic elements are *blocks* that offer a variety of functions and can be interconnected with *signals*. In Simulink, data can originate from constant blocks, function blocks that generate signals (e.g., a sine wave) or blocks representing an input from the environment such as from a sensor. Data flows between blocks via the signals, and blocks can process such data based on their function. Such functions include various mathematical operations (e.g., addition, comparisons, Boolean logic) and storing and loading data from *data stores*. More complex functions and operations are provided as well, such as determining derivatives, and can be constructed from basic blocks by means of specifying *subsystems*. The latter define a number of ports to which signals can be connected, and consist of other blocks, including (recursively) other Subsystem blocks.

For the storage of data, three different types of blocks are required: Blocks representing the data storage—Data Store Memory, blocks storing the input data—Data Store Write—and blocks reading the data and providing it as output—Data Store Read. The combination of these three types of blocks permits data to be buffered. Note that only the read and write blocks are connected with signals, the memory blocks are referenced by a parameter of the read and write blocks.

Further functionality can be provided directly as executable code (e.g., C or C++) which can be encoded into *system function*—S-Function—blocks. Such blocks are treated as black boxes that are given some input and based on this compute some output. This aspect is used by Stateflow [127], which describes state-based behavior of systems.

### 6.1.1 STATEFLOW

So far, the blocks that have been discussed model the flow of data, but do not describe behavior. For this purpose, *Stateflow* can be used, which is an extension for Simulink that is capable of modeling FSMs for discrete behavior, defined in *charts* (which are a specialization of Simulink S-Function blocks), similar to the statecharts of [74]. Simulink allows, similar to how modes, states and transitions are described in SLIM, discrete states and transitions between those states to be

specified. In addition, states can be composite, otherwise referred to as superstates, constructing them using substates.

Unlike SLIM, where each component has exactly one active mode or state, Stateflow supports two possible state types, depending on the configuration of the chart (or superstate): OR-states that are mutually exclusive, of which at most one is active, and AND-states, which are all active in parallel. Specifically, when a chart (or superstate) becomes active, either a single OR-state becomes active, or all AND-states become active. For the prior, the state which is activated is determined by the configuration of the chart (or superstate): Either a previously active state is reactivated, or the active state is determined by an incoming transition.

To interact with the surrounding Simulink model, Stateflow supports either the MATLAB or C language as the *action language*. At the occurrence of events such as entering or leaving a state, or taking a transition, actions can be executed that read or modify data. Furthermore, guards can be defined to control the flow in the chart. Data of the chart itself, such as the active state, is accessible to encompassing systems as well.

The transitions between states support the use of *junctions*, which allow multiple transitions to be connected to each other. When taking a transition leaving a state, such junctions must always lead to some target state eventually, however, multiple steps may be taken to do so. Simulink considers all transitions taken between two states as a single step, and therefore cannot be interrupted. If a junction has multiple outgoing transitions, the transition that is enabled with the highest precedence, according to a predefined order, is taken. This choice is always deterministic.

### 6.1.2 SIMULATION AND EXECUTION

One of the primary use cases of Simulink is the simulation of dynamic systems (as the name would allude to). For this purpose, various simulation engines are available. Such simulators work by resolving the dynamics of the system using discrete steps, resolving the system's ordinary differential equations (ODEs). Example solvers are based on the Dormand-Price method, or Runge-Kutta. However, various other approaches exist as well [128]. The major difference between the possible solvers lies in the size of the time steps. These can either be fixed a priori, or determined at each step based on how fast the variables in the system are changing, working toward minimizing the accumulated error over time.

Simulinks updates the state of blocks and signals according to a fixed order which honors the dependencies that (may) exist between blocks. Each block gets updated according to this order, subsystems updating recursively, which provides the updated signals that may be by subsequent blocks. This means that the dependency graph must be a DAG (or a set thereof), therefore prohibiting cycles in the model. However, as data stores are not connected directly, but are read/modified using data store read/write blocks, they may be used as buffers to break any potential cycles. This does, however, introduce a one step propagation delay of the signal.

An important caveat is that although Simulink can represent fully continuous behavior, its execution model inherently must approach this by discretization. However, in the scope of SLIM continuous behavior is limited to equations with a constant derivative, which can accurately and precisely be approached. Similar concerns also apply to, e.g., the SLIMSIM Monte-Carlo simulator (cf. Section 4.6).

When a Stateflow chart is executed, it will attempt to take the first possible transition with the highest priority. In particular, if two transitions are enabled, Stateflow will deterministically pick one, as opposed to SLIM which considers this a non-deterministic choice. Furthermore, if a transition is enabled over a period of time, if it is taken, it will be taken at the earliest possible time point.

This disparity is similar to that of using SMC for the analysis of SLIM models, as described in Chapter 4. Currently, the translator does not take any steps to resolve this. This means that the Simulink model may not consider all possible behavior that the SLIM model models, though any behavior that it does exhibit is sound with respect to the SLIM semantics (similar to what the Monte-Carlo simulator does). This difference is further discussed in Section 6.4.

## 6.2 SIMULINK TRANSLATOR

The translation from SLIM to Simulink is, for the most part, fairly straightforward. In particular the structure between the two modeling language is similar enough that the hierarchy can be retained. However, some semantical differences do exist, which need to be carefully considered and handled. The rest of this section will describe how the various aspects of a SLIM model are translated into a Simulink model.

The translation is performed on the SLIM instance model, recursively constructing it starting from the root component. Since Simulink supports recursive subsystems as blocks, this can be approached in a straightforward manner.

### 6.2.1 COMPONENTS

Components in the SLIM instance model are translated into subsystems in the Simulink model. Each such subsystems contains the blocks required to describe the (recursive) non-data subcomponents, data subcomponents, connections, and behavioral elements.

For each port, corresponding Inport/Outport blocks are added to the subsystem, which Simulink will make available to parent components. For event ports, no data type is set, whereas for data ports it is. To handle default values for data ports, a separate subsystem is added directly to this port, which contains a data store that holds either the default value, or the last written value. Its output then provides either the current value or, when the port is disconnected, the last value or default value. To allow resetting the stored value to default, a second input port is provided for this subsystem, connected to a block with constant value allowing it to reset

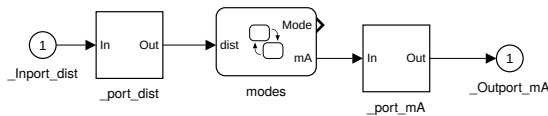


FIGURE 6.1 – Simulink representation of a subsystem with an in and out port, as well as the Stateflow chart for behavior.

the port to its default value. An example subsystem is shown in Figure 6.1. It also contains a Stateflow chart, which is further discussed in Section 6.2.3.

For data subcomponents, a data store memory is added, setting its type according to that of the data subcomponent. Read and write accesses to this data store are handled by the translations of the expressions found in the SLIM model.

The main exceptions to this are **clock** and **continuous** typed subcomponents, which change their value as time progresses. To handle these data types, aside from the memory store, an Integrator block is added, which accumulates the time spent since the last simulation step. The rate at which it does so is fixed to 1 for clocks, and multiplied with the rate of the active mode for continuous data components. An example of such a translation is shown in Figure 6.2. It involves three levels: One related to the individual trajectory expressions that are activated based on the current mode, followed by a subsystem that controls when the trajectory equation is active, with finally a subsystem containing the Integrator block responsible for updating the value of the **clock** or **continuous** variable.

For the root component, a subsystem is generated as well, labeled “slimModel”. The environment of the root component is then placed in the topmost Simulink system labeled “rootSystem”.

Each non-data component is furthermore associated with a Stateflow chart that describes its behavior. The behavioral aspects as determined by the modes and transitions will be described in Section 6.2.3. However, other elements are added to the Stateflow chart as well to represent the ports and data subcomponents. For each event port, a corresponding *event* object is added to the chart, which permits it to be used as a transition trigger. For data ports and data subcomponents, *data* objects are added to represent them. Similar to event objects, such data objects allow Stateflow transitions to access such data elements.

## 6.2.2 CONNECTIONS AND FLOWS

Simulink natively supports connections between ports, which are used directly by the translators. Two aspects need to be taken into account however: In the case of a fan-in connection, Simulink requires a Merge block to be used, for fan-out a branch point (the latter being a detail usually hidden from the user). For connections that can be disabled in certain modes, a block is placed in between that can enable and

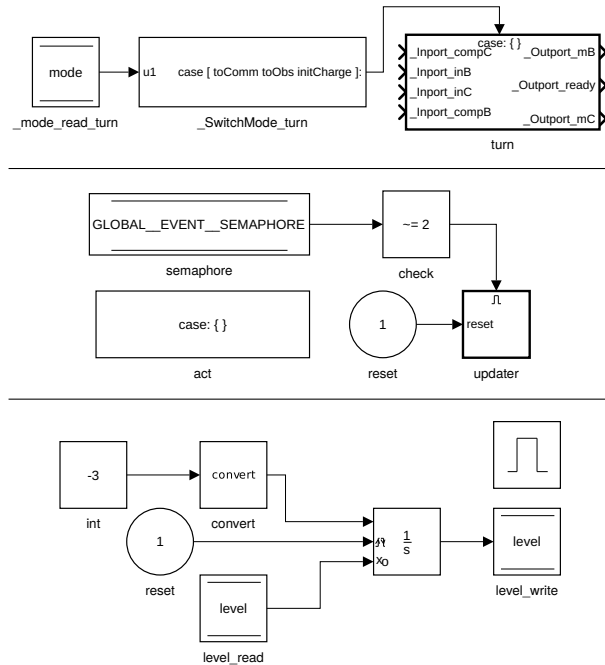


FIGURE 6.2 – Translation of continuous component in a system with two trajectory equations (top figure). The trajectory subsystem is active only when the global event semaphore (cf. Section 6.2.8) permits this, and can be reset (middle figure). An Integrator block reads the current value and updates it based on a constant rate (bottom figure).

disable the connection based on the active mode. This is explained in more detail in Section 6.2.5.

Flows are treated by constructing the blocks that represent the flow expression as will be described in Section 6.2.4, connecting the output of the last block in the expression to the target port of the flow.

One caveat, as already mentioned, is that connections that imply a cyclic dependency are prohibited. Such cycles may occur due to the use of data flows in SLIM. To break such cycles, such connections are buffered using a data store memory, as explained in Section 6.2.6.

### 6.2.3 MODES AND STATES

The modes and states of a SLIM component, as well as the transitions between them, comprise the behavior of the components. In Simulink, this can be expressed using Stateflow charts, which are a specific type of S-Function blocks (cf. Section 6.1.1).

SLIM modes and states are translated into Stateflow states directly. Invariants and

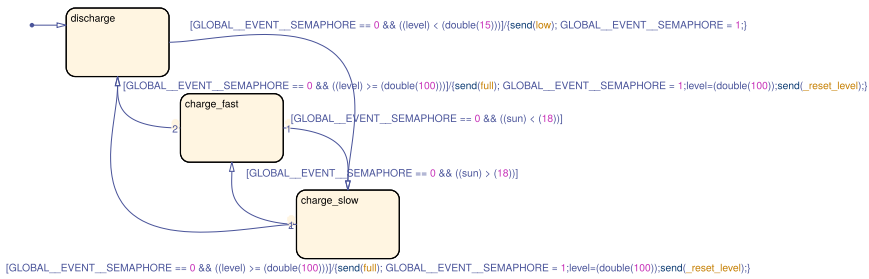


FIGURE 6.3 – Example Stateflow diagram. Three modes are included. The transitions contain guard used by the global event semaphore described in Section 6.2.8.

trajectory equations are not taken into account here, but are rather encoded using transition conditions in the Stateflow chart, and the Integrator blocks described in Section 6.2.1.

Mode and state transitions are translated into Stateflow transitions, using the mode/state to Stateflow state mapping described above to find the corresponding source and target states. The SLIM trigger, guard and effects are mapped to the Stateflow transition label. The transition label allows the triggering event, trigger conditions and transition actions to be specified. The event may refer to an input port of the chart, which permits the translator to map it directly. The condition is generated from the guard using the expression syntax that will be described in Section 6.2.4. Finally, effects are treated in a similar fashion as guards, but data elements are modified in the chart as well. An example is shown in Figure 6.3.

For the initial mode, a *default* transition is defined that points to the state that this mode is mapped to. This transition is, however, a special case and does not use any label.

The current mode of the chart is made available as an output port of the state chart. This port provides a value of an enumeration of possible state labels, which is used by other parts of the model that depends in the active mode, such as those influenced by dynamic reconfiguration.

Due to the deterministic behavior of Stateflow, mode invariants can be encoded as transition guards. Since a transition is taken as soon as it becomes enabled, no check is required if the time spent in a state exceeds what is allowed by the mode invariant. However, a mode invariant may still prohibit a transition from entering the corresponding state. Therefore, mode invariants are encoded as part of the guard using a conjunction.

#### 6.2.4 TYPES AND EXPRESSIONS

The types in SLIM, aside from the finitely enumerable ones, cannot directly be translated into Simulink, as Simulink can only represent types with a finite domain.

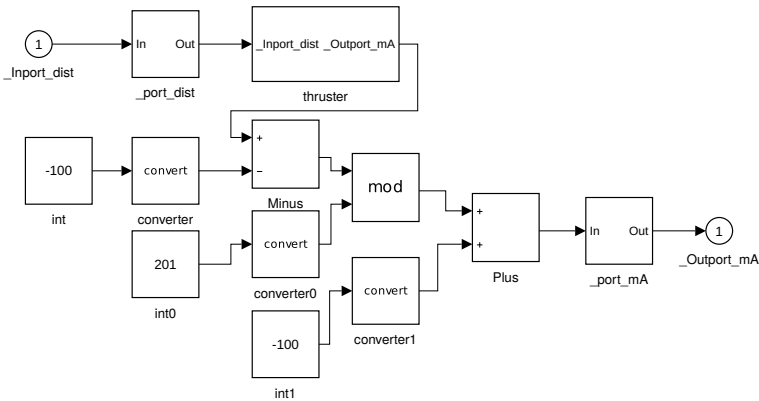


FIGURE 6.4 – Example flow expression as translated to Simulink blocks.

In such cases, in particular for integers and real numbers, the closest matching type is used.

Booleans are translated directly into the *bool* type. For enumerations, a new class needs to be defined for Simulink, which resides in a separate file. For each unique enumeration in SLIM, such a file is generated and then referenced. Ranges are translated to *int* with a fixed 32-bit width. This choice is made, ignoring the possibly lower width required for the range, to avoid the need to convert values when the range is promoted to an integer in a SLIM expression.

For integers, like ranges, a 32-bit wide *int* is used. This restricts the possible values, but cannot be avoided. If the model makes use of large integer values, it is recommended to configure Simulink to warn in the case of a detected overflow. For real numbers, a similar restriction applies. These are converted to the *double* floating-point type, which has a limited precision.

Expressions such as used by flows are expressed using various Simulink blocks. Simulink offers blocks for the various arithmetic and Boolean operations that correspond directly with those of SLIM. When a SLIM expression refers to an identifier for a port or data subcomponent, a corresponding read or write block is used to respectively read or write into the memory store representing this port or data subcomponent. The example shown in Figure 6.4 shows such a flow, which performs a modulus operation to wrap the input value in the range  $[-100..100]$ .

In Stateflow charts, expressions are translated into their equivalents in the C language, which is chosen as the action language by the translator. For mode transitions and effects, these expressions are used for the state transition labels, either as the transition condition (for guards), or the transition action (for effects). Stateflow allows such expressions to refer to data stores directly, so identifiers can be used as-is.

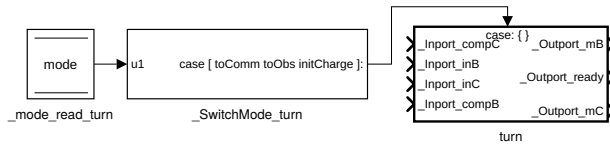


FIGURE 6.5 – Use of SwitchCase block to conditionally enable a Subsystem when an **in modes** specification is used.

One exception to this is the SLIM **case** operator, which has no direct equivalent in Simulink or Stateflow. If the result of the **case** expression is of Boolean type, it can be converted into a conjunction of implications. For other types however, this is not possible, and the translator therefore prohibits the use of such expressions.

### 6.2.5 RECONFIGURATIONS

A key feature of SLIM is its dynamic reconfiguration, which is expressed using the **in modes** specification. If this is used, the translator has to take a few extra steps to support this in the Simulink model. In particular, a connection has to be made between the active mode of a Stateflow chart, and the aspect of the model of which the configuration depends on this mode.

In general, mode-based activation makes use of a SwitchCase block that receives a mode value (based on a mode enumeration from a Stateflow chart) and outputs true if this mode corresponds to one specified by the **in modes** specification. This output can then be used to activate (or deactivate) part of the Simulink model, as shown in Figure 6.5.

For non-data subcomponents, a SwitchCase block is connected to the subsystem directly. Simulink supports enabling a subsystem by means of an ActionPort, which can be connected to the SwitchCase block corresponding to the active mode.

For connections, a subsystem is inserted between the source and target. This subsystem is then activated by means of a SwitchCase block. In addition, if a connection is disabled, a constant input is enabled representing the default value of the target port. A multiplexer block between the subsystem, default value and input port ensures the correct input is selected depending on whether or not the connection is active.

### 6.2.6 SIMULINK AND CYCLES

As mentioned previously, Simulink requires that a strict non-cyclic order between blocks is defined. However, a SLIM model may define data flows between two components in both directions, implying a mutual dependency. Direct data flow cycles are possible as well, if they are guaranteed to be broken by the mode configuration (using the **in modes** specification).

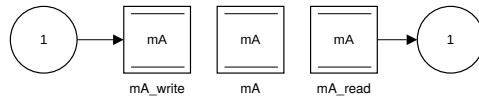


FIGURE 6.6 – Example subsystem used to buffer port data.

To permit such models to be translated to Simulink, two steps are taken in the translation process. First, any detected cycles are broken by replacing the signal representing the flow, or connection, by a data write and data read block, with corresponding data store, as depicted in Figure 6.6. The block evaluation priorities are adjusted such that the write will occur before the read.

A further consideration is that in SLIM, transition effects always evaluate variables according to the current state, and update them for the next state. As data stores are used to break cycles, it has to be ensured that value updates are still performed according to these semantics. For this reason, a global lock is introduced, labeled as `GLOBAL__EVENT__SEMAPHORE`, which enforces the correct order of updates. This is discussed in more detail in Section 6.2.8.

### 6.2.7 ERROR MODELS

The Simulink translator does not consider SLIM error models separately. Rather, it is used only after model extension is applied, which embeds the error model using fault injections in the nominal model.

However, model extension should not be applied when exporting to Simulink. The error model normally represents those errors possible in the system the Simulink model may be deployed to. Injecting it into the model would thus be redundant (and make the model more likely to fail), as both concrete errors and their modeled counterparts coexist.

Furthermore, the probabilities of error events are ignored, instead treating error events like other events. The translator does not support randomized behavior, which is something left for future work.

### 6.2.8 GLOBAL EVENT SEMAPHORE

A major difference in the semantics between SLIM and Simulink lies in the fact that SLIM considers synchronized transitions to occur atomically, whereas transitions/updates in Simulink occur sequentially. Thus, when in Simulink an output event is triggered, input events that should synchronize occur only if their priority ensures that they occur after the event. Furthermore, other output events may be triggered as well.

To force the SLIM semantics, a global event lock is added to the model. This event lock is used to force event synchronizations, and prevents time from pro-

gressing when an event is being executed. It is encoded as a data store labeled `GLOBAL__EVENT__SEMAPHORE` with an integer value ranging from zero to two.

The lock operates in three phases: In phase zero, the model is allowed to execute freely. In particular, at the beginning of a new simulation step, time is allowed to progress, as blocks related to time keeping are given the highest priority. Afterward, blocks are updated in order of their priority.

As soon as a chart triggers an output event, the phase is set to one. At this point, only transitions that receive this event are allowed to execute, and further updates of time are prohibited. The phase remains to be one until the simulation step ends, at which point it is set to two.

In phase two, the remaining transitions that receive the input event are executed. Any chart that already synchronized on this event is not updated. Thus, at the end of phase two it is ensured all transitions that synchronize on the input event triggered in phase zero have been executed. At this point, the phase is reset to zero.

### 6.2.9 LIMITATIONS

Not all the concepts and features of the SLIM language are supported by the translator. One of the major reasons is that while SLIM is a modeling language intended for the analysis of *possible* implementations, Simulink aims at modeling a *specific* implementation. This means in particular that any underspecification in the SLIM model cannot be handled as such by Simulink, but rather some specific concretization has to be made. This applies to race-conditions, non-deterministic transitions, possible time delays and random (error) events.

Other SLIM features that are not supported are non-Boolean **case** expressions, **@activation** transitions and event data ports. For these features no theoretical restrictions apply, but require extra implementation effort.

## 6.3 DEMONSTRATOR

A demonstrator was built in order to provide a more tangible representation of the capabilities of the COMPASS toolset. It allows a SLIM model to be deployed on a physical object, specifically a model of a satellite made using LEGO® and controlled using the LEGO® Mindstorms platform.

The SLIM model represents the various behaviors of the satellite. Its environment (sensors, actuators) is modeled using in- and output event and data ports on the root component. This SLIM model is then translated into Simulink. The Simulink model is extended by adding blocks for the various LEGO® Mindstorms devices. This extended model is used to generate code that can be executed by the Mindstorms platform. This code is then uploaded to the Satellite model, and executed.

The demonstrator models a simple space telescope, which consists of a satellite with payload — from hereon referred to simply as the satellite — that can be oriented

towards a particular point of interest, or in such a way that it can communicate with the ground station (where the two configurations generally exclude each other). Power is provided by solar panels and rechargeable batteries. The satellite therefore has the following main tasks:

- » perform observations, for which it requires some particular orientation,
- » communicate with ground, which requires a different orientation,
- » adjust for orbital decay, and
- » recharge batteries by pointing its solar panels towards the sun.

The observational capabilities (i.e., payload) are not modeled. Rather, the satellite has the task of orienting itself in a particular way such that an observation would be possible.

The orbital decay is modeled by lowering the altitude of the satellite, which can be measured. Enabling the thrusters will increase the altitude again. This function applies hysteresis to adjust the orbit when some lower bound has been crossed. The satellite is then lifted until some upper bound has been reached, at which point it is allowed to (slowly) decay again.

Recharging the batteries is modeled by adjusting the orientation of the solar panels in such a way that they receive the maximum amount of light. If the input is larger than some predefined threshold, the batteries are considered charging, otherwise discharging. The charge and discharge rates, as well as the battery level, are modeled artificially.

The environment of the satellite is modeled as having various sensors and actuators. A height sensor is available to report the height of the satellite above ground, and a light sensor to report the intensity of the light hitting the solar panels. To move the satellite, three motors are used. One increases the height of the satellite ( $m_{thrust}$ ), one can adjust its orientation ( $m_{turn}$ ) and finally one is used to adjust the solar panels ( $m_{solar}$ ).

### 6.3.1 SLIM MODEL

The SLIM model is structured as follows: The main component of the specification, and also the root component of the model instance, is the **system** Sat. It has two input data ports that provide the values from the height and light sensors, three output data ports to drive the motors, and two input data ports to read the position from the  $m_{turn}$  and  $m_{solar}$  motors.

It operates in either of the *observing*, *communicating* or *charging* modes, with transitional modes in between. In the observing mode, the satellite is oriented such that it can make its observations according to its mission, in particular away from ground. In the communicating mode, instead it is oriented to ground to allow radio communications. Finally, in the charging mode it orients itself and the solar

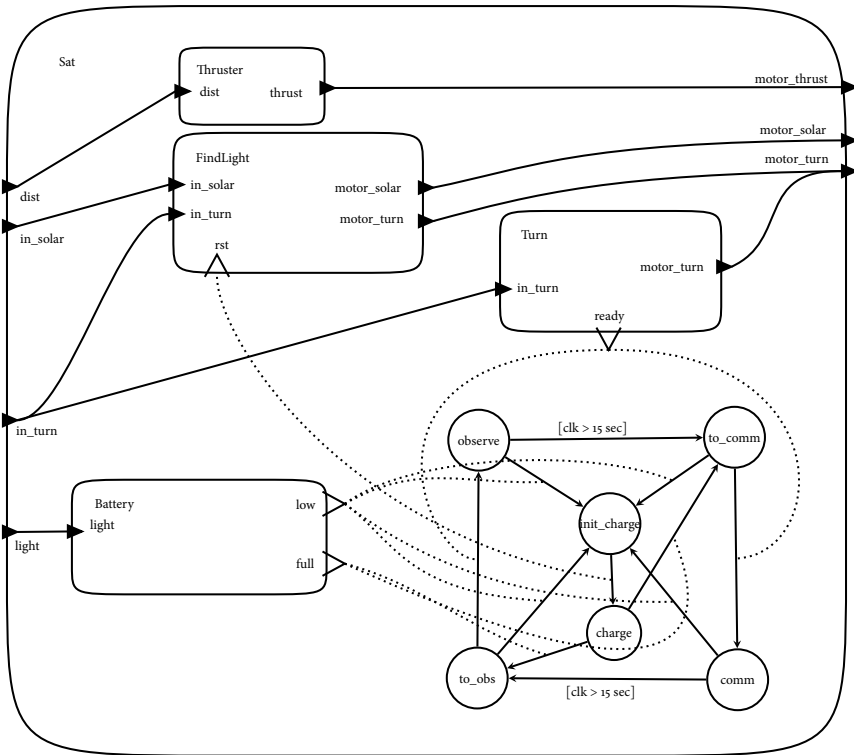


FIGURE 6.7 – Overview of the SLIM model of the satellite.

panels such that they receive the most light to recharge the batteries. In this mode, communications and observations are halted to conserve power.

The Sat component is constructed from four different subcomponents, which are:

- » the thruster system,
- » the battery,
- » the orientation control, and
- » the solar panel control.

An overview is shown in Figure 6.7.

The thruster system (Thruster) reads the current height, and based on this measurement either switches on or off the corresponding motor, applying hysteresis. It has two modes, up and down, which set the motor speed (which when not moving up is set to move slowly downwards as to model the decay of the orbit), and transitions in between that get triggered upon reaching a certain height.

The battery system (Battery) models a battery with a charge level that ranges between 0 and 100, an input port light to measure the input power, and two event

ports to indicate the battery being full or low on power. Internally, it operates in either of three modes: `discharging`, `charge_slow` when power input is low, and `charge_fast` when input power is high. All three modes have a (dis)charge rate associated as part of the mode invariant. Transitions between the different modes are guarded according to the level of input power, and send the `full` or `low` events when corresponding charge levels are reached.

The orientation control sets the orientation of the satellite according to parameters that are given by the main satellite system. It rotates the satellite until this orientation has been attained by comparing it against the values received from the motors. When the desired orientation changes it will reset and rotate the satellite again.

When the satellite switches to the charging mode, the component responsible for adjusting the solar panels, `FindLight`, is given control over the motors. The component simply rotates the satellite and solar panels until the maximum value of light is registered from the solar panels. It does so by first rotating the satellite until a (local) maximum is detected. Then the solar panels are rotated until again a (local) maximum is detected. The satellite will then remain in this orientation until the batteries are full again.

The charging, communicating and observing modes are mutually exclusive, meaning the motors and sensors for the orientation and solar panels are controlled by one system at a time. The thrusters and battery subsystems operate independently of the mode of the system. These are also considered critical as the failure of either results in the loss of the satellite.

### 6.3.2 LEGO® MODEL

The LEGO® model of the satellite itself is constructed as a cube-shaped frame that contains the various actuators that allow it to move, see Figure 6.8. It is attached to a pole with a rack, to which a pinion gear is attached that is part of the satellite, driven by motor  $m_{thrust}$ , allowing vertical movement. The satellite is also allowed to rotate around the pole for orientation, driven by motor  $m_{turn}$ . Attached to the sides of the satellite are two solar panels, which can be rotated along their longitudinal axis by motor  $m_{solar}$ .

An ultrasonic sensor is positioned underneath the satellite, permitting the satellite to measure the distance between itself and the ground plate. The solar panels provide a voltage that depends on the received light intensity, which can be used as a measure of received power. The motors are also capable of providing the degrees of rotation that they have been turned, either when driven, or by external force.

#### *LEGO® Mindstorms*

To control the satellite, the LEGO® Mindstorms platform is used, in particular the EV3 variant. A single controller, also referred to as a “brick”, can receive data from up to four sensors, and drive up to four actuators. A program can be transferred to the controller a priori, via USB or Ethernet, which can then be run standalone.

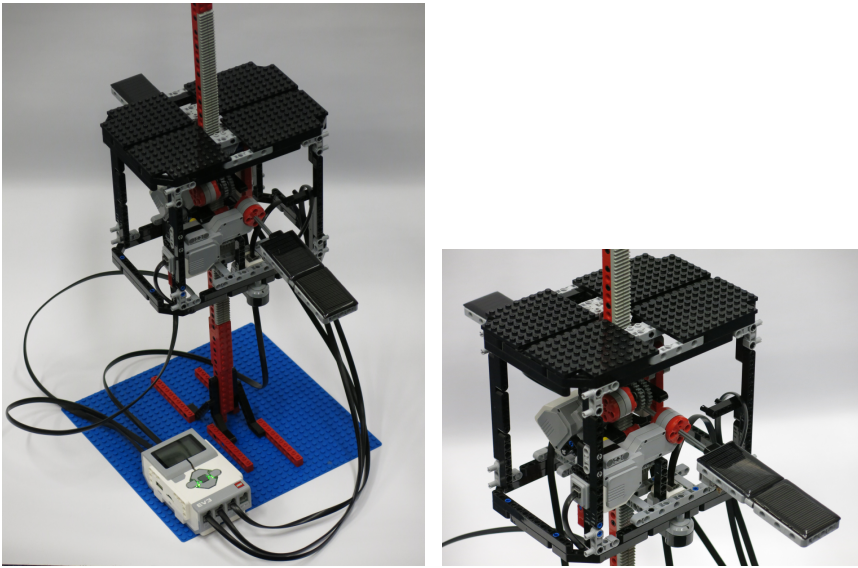


FIGURE 6.8 – Overview and closeup of the LEGO® satellite model.

The motors and ultrasonic sensor are provided as part of the EV3 kit, and can directly be used with the controller. For the solar panels, a legacy interface is used which permits the use of an internal analog to digital converter (ADC) to measure the input voltage. This requires a modification to the drivers used by the Simulink translator to read this value directly, as normally it is compared to a threshold to provide a Boolean value. The motors and sensors of the satellite are connected to the controller directly, which is placed on the ground due to its weight.

### 6.3.3 DEPLOYMENT

Most of the translation between the SLIM model and the LEGO® model is done automatically. However, some steps have to be taken manually and some modifications to the Simulink model are required for it to be complete. An overview of the entire flow is depicted in Figure 6.9.

To translate SLIM into Simulink, the COMPASS toolset is used. For this a CLI script is available that can be given the SLIM model, and will generate the necessary Simulink file(s).

In order to interface with the Mindstorms motors and sensors, blocks need to be added from the Mindstorms library available for Simulink. This needs to be done by hand, as the translator has no provisions for this. These blocks can be connected to the ports provided by the Subsystem representing the translated SLIM model, and configured to map to the physical ports of the Mindstorms controller. There

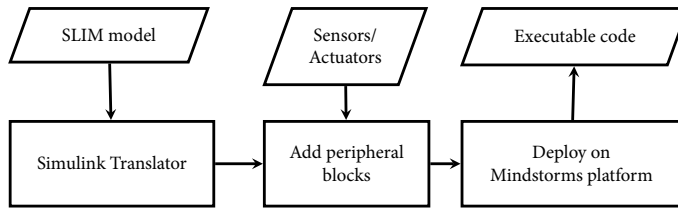


FIGURE 6.9 – Graphical representation of the process used to generate code from a SLIM model.

are blocks for the various types of Mindstorms elements, including the used motors and sensors.

The Simulink model can be deployed to the Mindstorms controller from MATLAB directly. With the controller connected via USB or the Ethernet, the code generated by MATLAB can be uploaded automatically, and then executed either from MATLAB itself, or by running it from the controller directly.

## 6.4 DISCUSSION

The work presented in this chapter was primarily performed in order to make it possible to give a more concrete representation of what the COMPASS toolset is capable of doing. The use of LEGO® to construct the model allows it to be as flexible as the modeling language used, meaning it is fairly easy to make changes as the need arises.

Using Simulink as an intermediary language comes with a number of benefits. It is a well-known language with a significant adoption rate in industry. Furthermore, and particularly relevant for the application of the demonstrator, the Simulink toolset supports automatic code generation capable of targeting the Mindstorms platform.

However, Simulink is not fully compatible with SLIM, and some disparities exist between the two languages. In particular, the differences in handling event synchronization, evaluation order and non-determinism stand out. For the first two, workarounds are possible, though these tend to complicate the translated model.

Non-determinism in the SLIM model is not handled explicitly. The resulting implementation is *necessarily* deterministic, however, the choice of transition or time delay is not controllable. The translation remains sound w.r.t. the original SLIM model however. Arguably, such models should be prohibited, or could for demonstration purposes be randomized.

Nevertheless, the translator is capable enough to provide a complete workflow starting from a SLIM model for a satellite to a working implementation on the LEGO® Mindstorms controller. By doing so, it has been shown how the abstract constructs of the modeling language can be interpreted in real world applications.

#### 6.4.1 FUTURE WORK

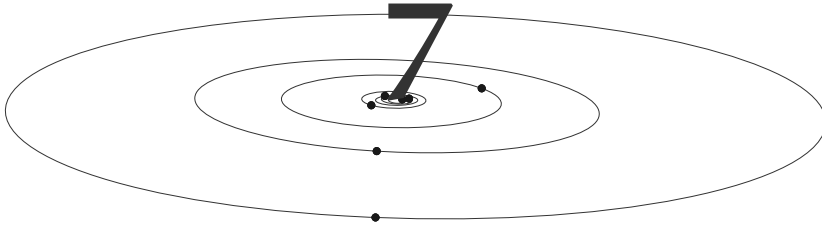
Although the current implementation is sufficient for its purpose, improvements are possible, a few of which are mentioned here.

At the modeling level, the translator can be made more feature complete, supporting more of the SLIM modeling constructs. In particular, **case** expressions, **@activation** transitions, **event data** ports and probabilistic error events are not (fully) supported.

Internally, the translator makes use of the text-based MDL file format for Simulink. However, this format is nearing deprecation, and to be replaced by the SLX format, which is an XML-based format. Although the structure of the compressed SLX is defined by the Open Packaging Conventions (OPC) [OPC], the specification of the embedded files, like for MDL files, is not open and requires significant effort to be understood.

Finally, it would be interesting to build a complete case study around the approach, including applying the various analysis capabilities of the COMPASS toolset on the model. Alternatively, the same approach can be tried in the case study discussed in Chapter 7. Furthermore, it may be interesting to complement the translation with a model-based testing approach, allowing a running implementation to be tested against a SLIM model.





## MODELING AND ANALYSIS OF THE CUBETH NANO-SATELLITE IN SLIM

*ABSTRACT – Based on a prior case study where a nano-satellite was designed using the BIP language, a case study was also performed on the same design making use of the SLIM language and the COMPASS toolset. The aim of this case study is to provide an openly accessible model, as well as determine how well the COMPASS toolset can handle such models. First, the architectural patterns that were used in the BIP-based model are defined for SLIM, followed by the properties that can be derived from these patterns. The architectural patterns are subsequently applied to specify the complete SLIM model, and the properties used in the BIP-based case study are translated as well. These were subsequently analyzed using the COMPASS toolset.*



USUALLY, the design of satellites is subject to strict constraints, not in the least with regard to its weight. It is therefore no surprise that there has been an increasing demand for launching small satellites, i.e., satellites with a mass below 500 kg. A particular subset of those, the nano- and pico-satellites, saw an increased number of launches in the past few years. Such satellites are light — less than 10 kg [91] — and generally cost significantly less than larger satellites. Such satellites can benefit from the ability to piggyback on launch vehicles used to bring larger satellites into orbit, though launch vehicles designed specifically for small satellites exist as well [99].

For pico- and nano-satellites, the CUBESAT [41] standard can be employed, which is a standard designed specifically to permit multiple satellites to share a single *deployer*, allowing them to share a ride on a multitude of possible launchers [108]. The physical dimensions of such satellites are limited to a width and height of 10 cm, and a length in units of 10 cm, e.g., a length of 10 cm being 1U, and a length of 30 cm

being 3U.

To further reduce costs, commercial off-the-shelf (COTS) hardware may be used, as well as simple communication buses such as inter-integrated circuit (I<sup>2</sup>C). Such hardware is, however, less resilient to the harsh environment of outer space, in particular when it comes to radiation. In that case, the software (or firmware) running on the satellite will have to implement additional error detection and correction mechanisms to deal with this. This leads to a stronger emphasis on software reliability, and increases its complexity as well, stressing the importance of software validation and verification.

This chapter will present a case study based on the CubETH project [81], in which a nano satellite is developed with the objective of determining its orbital position using COTS global navigation satellite system (GNSS) hardware. As part of the project, its software architecture was modeled [115] using the BIP language [20]. Based on this architecture, a case study was performed in which various architectural patterns were defined and analyzed [102, 103].

The same case study was performed using the COMPASS toolset. The main goal is to provide a larger case study for which the sources are readily accessible (those of a prior case study [60] are unfortunately not available), as well as being able to compare both the BIP framework and the COMPASS toolset.

The remainder of this chapter will discuss the original approach, and how it compares to the SLIM version. First, a brief introduction to the CubETH project and the BIP framework is given. The architectural patterns from [103] are re-constructed in SLIM, followed by the model of the architecture itself. Finally, similar to the original case study, deadlock analysis and model verification are performed—the latter making use of both properties derived from the architectural patterns, as well as those derived from requirement specifications.

## 7.1 CUBETH PROJECT

One particular project making use of the CUBESAT platform is CubETH [81], which aims to develop a satellite which is capable of accurately determining its position and attitude in orbit, using low-cost hardware. In order to validate its measurements, it features laser-ranging reflectors to determine its position from the Earth's surface.

The software architecture of the CubETH project has been described with the BIP language [21]. Originally, the architecture was specified in [115]. Later, from this specification, various architectural patterns were identified that provide reusable structures for which certain properties can be proven to hold [102]. These patterns were subsequently used to re-construct the architecture, ensuring that the composed system fulfills the properties defined by the patterns *by construction*.

The original case study has been performed using the BIP framework [15], which provides capabilities similar to those of COMPASS. The BIP language, from hereon referred to as BIP, is structurally similar to AADL (and thus SLIM), making it interesting to perform this case study with the COMPASS toolset.

In BIP, a hierarchical structure is modeled using *atomic* and *composite* types. These notions are similar to those used by AADL, though in the case of BIP the implementation of these types differ (whereas for AADL the difference is simply the absence or presence of subcomponents).

Atomic components, created with the **atomic type** keyword, specify ports, states and transitions in a fashion similar to SLIM. Ports are declared with a given port type that has to be defined beforehand. Such types can be empty for simple interactions, or some positive number of typed variables for data communication. This is similar to the event and event data ports in SLIM (not data ports). Ports in atomic components can be either internal, or external using the **export** keyword. The latter permit synchronization, the prior are for internal events only.

As with SLIM, atomic components can specify data variables and clocks, which function in a similar fashion. The case study makes use of discrete variables only, therefore clocks will not be considered further. Although some parts of the system refer to timed operations, these are abstracted into non-deterministically triggered events, allowing the time period to remain unspecified.

The behavior of an atomic component models a Petri net [21], specified by means of states, transitions, and an initial transition. The states and transitions function in a fashion similar to SLIM, with the possibility to specify guards and effects for each transition. The initial transition in particular can have effects specified that determine the initial value of variables (which in SLIM is done with the `Default` property).

Compound component specifications consist of subcomponent definitions and connectors which provide the connections between the various ports. Subcomponent definitions work in the same fashion as SLIM, specifying the type of the component. Connectors determine how the ports of the components in the hierarchy may interact [26]. Whereas the behavior of the system is determined by the Petri net that is defined for atomic components, the connectors of compound components determine the interaction and priority aspects. A connector specifies how some source port communicates what a (possibly empty) set of receiving ports. Connectors can be specified such that they require all receiving ports to synchronize by marking the source port as a *synchron*, or on any number of subsets of those (including the empty set), individually referred to as an interaction, by specifying the source port to be a *trigger*. Of these options, the first — specifying rendezvous

semantics — corresponds to the current semantics of SLIM<sup>1</sup>. This is the only type of connector used in this case study.

Priorities can be specified between the possible interactions of the connectors in a component. In addition, BIP will automatically specify priorities according to the maximal progress rule that favors interactions with the highest number of participating ports. SLIM has no direct counterpart to this, instead requiring the use of either dynamic reconfiguration, or transition guards.

Clearly, the main difference between BIP and SLIM lies in the possibilities for the specification of interactions and priorities. SLIM has no direct counterpart for priority specifications — the choice between multiple possible interactions is always made non-deterministically. Different possible interactions can to some degree be emulated by using mode-dependent connections. However, the difference between rendezvous and broadcast interactions cannot be made. How these difference are tackled for the case study is described below.

### 7.2.1 FROM BIP MODELS TO SLIM MODELS

Most of the components of a BIP specification have a direct counterpart in SLIM, and so translation is fairly straightforward. A quick comparison of the two languages can be found in Table 7.1. The following describes in detail how the BIP models have been translated to SLIM.

Within the case study, one type of port, in the example of Listing 7.1 shown as `syncPort`, is used, which is a simple untyped port which may be directly translated to an event port in SLIM.

For each atomic component type, a new instance of a system component is created in SLIM, with each port mapped to an event port. The direction of the port depends on the semantics of the model. Depending on the architectural pattern used (as will be discussed in Section 7.3), such a port can be mapped to either an input or output port. In general, however, when a port is used as a source port in a connector, it can be considered an output port, and an input port otherwise.

Every other element of a BIP component is translated into some element of a new SLIM system component implementation. Data components are translated directly into data subcomponents. The default values are set according to the assignments that may be found in the initial transition of the component's behavior.

Regarding the behavior, each place of an atomic component is translated into a state in the SLIM component, and transitions are mapped accordingly. Guards are transliterated, as are the effects. The exception is the initial transition: As mentioned, the effects of this transition determine the default value of data subcomponents. Its target place is marked as the initial state in the SLIM model (note that for

---

<sup>1</sup>Prior version of SLIM used the broadcast semantics, though the minimum required number of receiving ports was one.

TABLE 7.1 – Overview of differences and correspondences between BIP and SLIM.

BIP	SLIM
port type	event (data) port
synchron connector (with interactions)	port connections (mode-based)
trigger connector	-
atomic/compound type	component implementation
(export) port	event (data) port
initial transition	initial mode
data	data subcomponent
component	non-data subcomponent
connector	port connection
places	modes/states
priorities	-

BIP there is no difference between an initial place and activation place, as it does not have the reactivation semantics from SLIM).

For compound components, each subcomponent is mapped directly to a subcomponent of the corresponding component implementation in SLIM. More effort is required for the connectors: The connectors are all defined as rendezvous, with the number of receiving ports ranging from zero to eight. Within the case study they are used in three ways:

- » A single interaction between multiple ports;
- » Multiple connectors where one or more ports are not required to rendezvous, effectively emulating a connector with a trigger source and multiple interactions with a common subset;
- » A single port which may trigger at any time.

Clearly, the first type of usage can be directly translated into connections between the source port and each receiving port. The second type can be transformed similarly, though with a caveat: The optional port must be defined as non-blocking to permit synchronization to occur between the other ports. SLIM does so by making a port which is non-blocking input enabled in all states or modes. The third case can be represented as an unconnected output port, which will never block, or an input-enabled input port.

### 7.3 ARCHITECTURE PATTERNS

The CubETH case study [102] provides a number of architectural patterns (or styles) for BIP models, that can be used to implement recurring patterns. These patterns apply to SLIM models as well. However, some modifications are required to provide the same behavioral properties under the SLIM semantics. Other modifications can be made to simplify the construction as well (due to the availability of different

```

port type syncPort

connector type SINGLE(syncPort p1)
    define p1
end

connector type RDV2(syncPort p1, syncPort p2)
    define p1 p2
end

atomic type WatchdogReset
    data int timer
    export port syncPort internal_watchdog
    export port syncPort done

    place S0, S1

    initial to S0 do{ timer=0; }

    on internal_watchdog from S0 to S1 do { timer = 0; }
    on done from S1 to S0
end

compound type CdmsStatus
    component WatchdogReset WATCHDOG
    component CdmsStatusActionFlow CDMSSSTATACTFL
    component MessageLibrary MSGLIB

    connector RDV2 internal_watchdog (WATCHDOG.internal_watchdog,
                                        CDMSSSTATACTFL.start_reset)
    connector RDV2 reset_done (CDMSSSTATACTFL.done_reset,
                                WATCHDOG.done)
    connector RDV2 cdms_I2C_res (CDMSSSTATACTFL.I2C_res,
                                  MSGLIB.decodeMessage)
    connector RDV2 cdms_I2C_ask (CDMSSSTATACTFL.I2C_ask,
                                  MSGLIB.composeMessage)
    connector SINGLE CDMSSSTATACTFL_start (CDMSSSTATACTFL.start)
    connector SINGLE CDMSSSTATACTFL_finish (CDMSSSTATACTFL.finish)
end

```

Listing 7.1: Example type specifications in BIP. WatchdogReset is an **atomic** type, CdmsStatus is **compound**.

```

system WatchdogReset
features
    internal_watchdog : in event port;
    done : out event port;
end WatchdogReset;

system implementation WatchdogReset.I
subcomponents
    timer : data int {Default=>"0"};
states
    S0: initial state;
    S1: state;
transitions
    S0 -[internal_watchdog then timer := 0]-> S1;
    S1 -[done]-> S0;
end WatchdogReset.I;

system CdmsStatus
features
    i2c_ask : out event port;
    i2c_res : in event port;
end CdmsStatus;

system implementation CdmsStatus.I
subcomponents
    watchdog : system WatchdogReset.I;
    controller : system CdmsStatusActionFlow.I;
modes
    m0 : initial mode;
transitions
    m0 -[ controller.start or controller.finish ]-> m0;
connections
    internal_watchdog : port controller.start_reset ->
                        watchdog.internal_watchdog;
    reset_done : port watchdog.done ->
                controller.done_reset;

    cdms_i2c_ask : port controller.i2c_ask -> i2c_ask;
    cdms_i2c_res : port i2c_res -> controller.i2c_res;
end CdmsStatus.I;

```

Listing 7.2: Components from Listing 7.1 converted into SLIM.

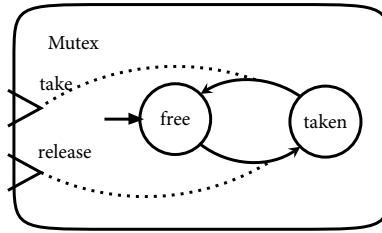


FIGURE 7.1 – Structure of the mutex pattern.

modeling constructs). The following sections will provide the SLIM versions of these architectural patterns.

Each architectural pattern consists of one or more *operand* components. In a few cases, one central component can be designated as the *operator* or *managing* component. This will be explicitly mentioned for each pattern where applicable.

For each architectural pattern, both assumptions and guarantees are defined, referred to as *assumed* and *characteristic* properties in the original case study [102]. The assumptions place restrictions on the behavior of the operand components in order to be able to provide the desired guarantees. Effectively, they require these components to be well behaved. The guarantees then satisfy the architecture's requirements.

In summary, the following patterns will be considered:

- » Mutual Exclusion
- » Client-Server
- » Action Flow, with optional abort
- » Failure Monitoring
- » Mode Management
- » Buffer Management
- » Event Monitoring
- » Priority Management

### 7.3.1 MUTUAL EXCLUSION

The *mutual exclusion pattern* is used to ensure that only one component can perform a given task at any time, or make use of some shared resource. The pattern realizes this by only permitting a single component to synchronize on an action that enters the critical section of the mutex, and requires it to release it before another component (or the same) can acquire it again.

The patterns consists of one managing component, from hereon referred to as the mutex component, that models the critical section, and any number of operand components. The mutex component has two *blocking* input event ports, take and release which respectively take and release the exclusive right to use it.

The pattern realized the desired properties by having the mutex component act and block on input event ports. As SLIM permits only one component to trigger an

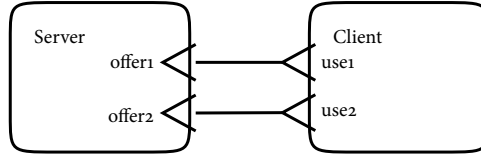


FIGURE 7.2 – Structure of the client-server pattern.

event at any given time, only one component is capable of synchronizing with the take event. Similar constraints hold for the release event.

The assumptions require that the client components always release a taken resource eventually, and only alternate between the take and release events (i.e., multiple subsequent uses of take or release are disallowed). Formally:

$$\begin{aligned} & \square(\text{begin\_task} \rightarrow \diamond \text{end\_task}) \wedge \\ & \square(\text{begin\_task} \rightarrow \circ(\neg \text{begin\_task} \mathcal{U} \text{end\_task})) \wedge \\ & \square(\text{end\_task} \rightarrow \circ(\neg \text{end\_task} \mathcal{W} \text{begin\_task})) \end{aligned}$$

The mutex pattern then guarantees only one client can begin its task when the mutex is free, and all other must wait until it is released. Formally, it suffices to ensure that

$$\square(\text{take} \rightarrow \circ(\neg \text{take} \mathcal{U} \text{release}))$$

### 7.3.2 CLIENT-SERVER

The *client-server pattern* ensures that one client can make use of a service provided by a server at a time, though the server may provide more than one service, and multiple clients may use those services concurrently.

Unlike the mutual-exclusion pattern, where there critical resource can be held for some period of time, the use of a service is considered a single (transactional) event. In SLIM, this is trivially realized by associating a single input event port with each service provided by the server component, to which any number of clients may connect. As with the mutual exclusion pattern, sharing a single input event port allows only one client to synchronize at any given time.

No assumptions are required on the behavior of either the clients or the server. The guarantee that only one client can make use of a single service at a time can be formalized as follows:

$$\forall i \in [1..n]. \square(\text{client}_i.\text{use} \rightarrow \bigwedge_{j \in [1..n], j \neq i} \neg \text{client}_j.\text{use})$$

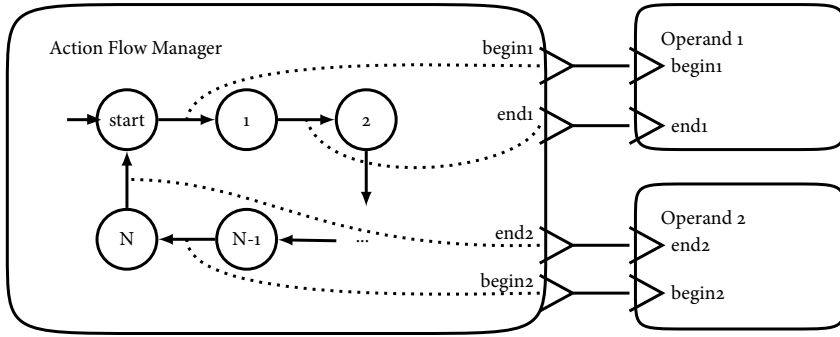


FIGURE 7.3 – Structure of the action flow pattern.

### 7.3.3 ACTION FLOW

A common type of requirement is that a certain sequence of actions has to be adhered to. The *action flow pattern* ensures that this is the case by a specific ordering of system modes, effectively constructing a state machine. The pattern consists of a flow manager component, and any positive number of operand client components. The manager features two *output* event ports for each action, one to indicate its start and one for its end. Each client component has matching *blocking* input event ports. Upon receiving the event matching the begin of an action, it will begin performing that action. When done, it maintains a state in which it is ready to receive the action end event. When all client components are ready to receive such an event, the manager component will trigger it.

This approach ensures each client performs the same sequence of actions at the same time (though possibly with different execution times). When synchronization between clients is not necessary or desired, multiple manager components can be used.

Assumptions require correct behavior from the clients, in particular that each begin of an action is followed by its end. That is, for a given operand component *client* and *m* actions for this operand:

$$\forall a \in [1..m]. \square(client.begin_a \rightarrow \diamond client.end_a)$$

The pattern then guarantees that the operands execute the possible actions according to the order of the flow, and only continue to the next action when the previous has been completed:

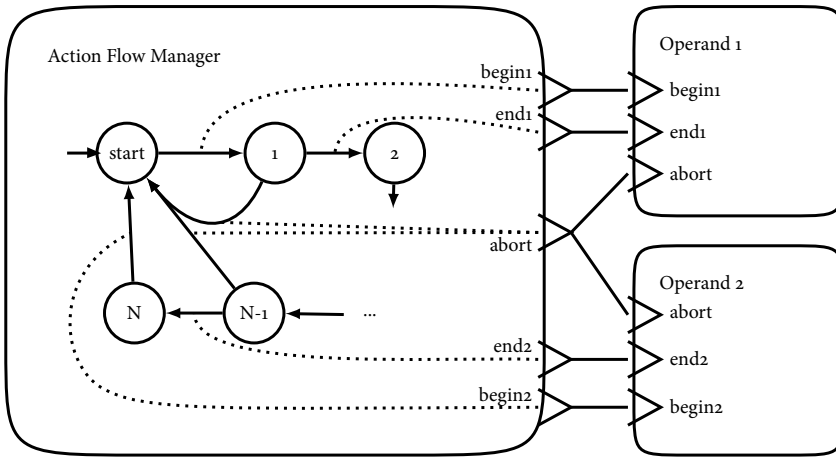


FIGURE 7.4 – Structure of the action flow with abort pattern.

$$\begin{aligned}
 & \forall a \in [1..m]. \\
 & \square(\text{begin}_a \rightarrow \circ(\neg \text{begin}_a \mathcal{U} \text{end}_a)) \wedge \\
 & \square(\text{end}_a \rightarrow \circ(\neg \text{end}_a \mathcal{W} \text{begin}_a)) \wedge \\
 & \square(\text{begin}_a \rightarrow (\neg \text{end}_a \mathcal{W} \text{begin}_a)) \wedge \\
 & \square(\text{begin}_a \rightarrow \circ(\bigwedge_{a' \in [1..m]} \neg \text{begin}_{a'}) \mathcal{U} \text{end}_a) \wedge \\
 & \square(\text{begin}_a \rightarrow \circ(\bigwedge_{a' \in [1..m]} \neg \text{begin}_{a'} \mathcal{W} \text{begin}_{a+1}))
 \end{aligned}$$

The semantics of SLIM ensure all clients synchronize on the **begin** and **end** events, due to the use of output events on the managing component.

#### Action Flow with Abort

As an extension to the action flow patterns, the *action flow with abort pattern* adds the ability to abort the flow during any action, which restarts it. This is realized by additional transitions in the flow manager from the modes in which an action is running to the initial mode with a separate abort action.

Aside from the assumptions and guarantees of the action flow pattern, the abort operation resets the order of actions to be performed to the initial action regardless of the current one. This requires that all operand components have this port, and it is input enabled. As the client is allowed to perform an abort between a begin and end action, the assumptions become:

$$\forall a \in [1..m]. \square(\text{client.begin}_a \rightarrow \diamond(\text{client.end}_a \vee \text{client.abort}))$$

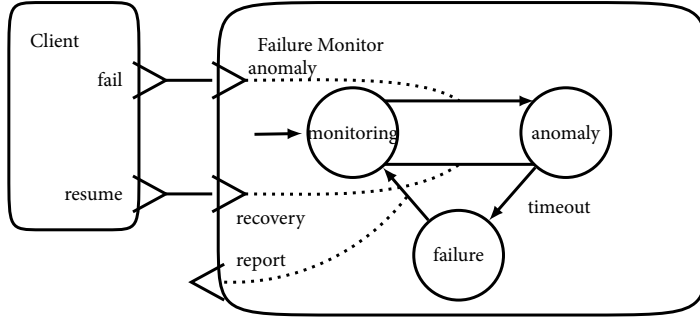


FIGURE 7.5 – Structure of the failure monitor pattern.

The pattern then guarantees the following:

$$\begin{aligned}
 & \forall a \in [1..m]. \\
 & \square(\text{begin}_a \rightarrow \circ(\neg\text{begin}_a \mathcal{U} \text{end}_a)) \wedge \\
 & \square(\text{end}_a \rightarrow \circ(\neg\text{end}_a \mathcal{W} \text{begin}_a)) \wedge \\
 & \square(\text{begin}_a \rightarrow \circ(\bigwedge_{a' \in [1..m]} \neg\text{begin}_{a'}) \mathcal{U} (\text{end}_a \vee \text{abort})) \wedge \\
 & \square(\text{begin}_a \rightarrow \circ(\bigwedge_{a' \in [1..m]} \neg\text{begin}_{a'}) \mathcal{W} (\text{begin}_{a+1} \vee \text{abort})) \wedge \\
 & \square(\text{abort} \rightarrow \circ(\bigwedge_{a' \in [1..m]} \neg\text{begin}_{a'}) \mathcal{W} \text{begin}_1)
 \end{aligned}$$

The SLIM semantics ensure that, in addition to what was stated for the plain action flow pattern, all clients synchronize on the abort action, which in turn ensures the system remains in a consistent state.

#### 7.3.4 FAILURE MONITORING

The *failure monitoring pattern* provides a component which monitors some operand for anomalies, and reports them when they appear to not be transient. When the operand component sends a signal indicating failure, the operator component enters a state that indicates an anomaly has been detected. When the operand does not send a recovery signal within some time bound (which may be zero)<sup>2</sup>, the operator will enter a state indicating a failure, and report. After this, it will return to the monitoring state, allowing the same operations to be performed again. Both the failure and recovery signals are input enabled to ensure the failure monitor does not influence the behavior of the operand component.

Note that the original case study labels the monitoring state NOMINAL, following the requirements specified for the CubETH subsystems (see Section 7.4.1). This is slightly misleading in this context, and caused by the fact that the states of the

<sup>2</sup>For the case study, it is assumed unspecified and is modeled as a discrete, non-deterministic event.

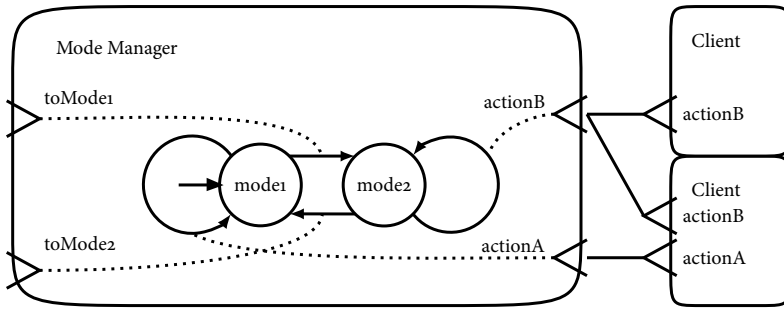


FIGURE 7.6 – Structure of the mode management pattern.

failure monitoring pattern have a different scope than those considered by the requirements.

The pattern does not require any assumptions from the client. It guarantees that the anomaly is reported after a timeout if no recovery happened beforehand:

$$\begin{aligned}
 & \square(\text{anomaly} \rightarrow \diamond(\text{failure} \vee \text{recovery})) \wedge \\
 & \neg\text{failure} \mathcal{W} \text{anomaly} \wedge \\
 & \square(\text{recovery} \rightarrow \neg\text{failure} \mathcal{W} \text{anomaly}) \wedge \\
 & \square(\text{failure} \rightarrow \neg\text{recovery} \mathcal{U} \text{report}) \wedge \\
 & \square(\text{failure} \rightarrow \neg\text{failure} \mathcal{U} \text{recovery})
 \end{aligned}$$

### 7.3.5 MODE MANAGEMENT

The *mode management pattern* is used to determine what set of possible actions is enabled for some operand components based on the active mode of the operator component. Which mode is active is determined by a set of operand components that trigger mode switches.

The operator component defines a state machine, where each state determines the allowed actions. In SLIM, this is realized using modes with self-loops synchronizing on the actions, which are modeled as input events. Transitions between modes are triggered by input events connected to operand components that control the active mode. The events controlling the actions are connected to the operand components that potentially perform these actions.

No specific assumptions are required from operator components. The pattern guarantees that operand components can only perform those actions that the manager permits in a certain mode. Given a function  $\text{Acc} : M \times \text{Act}$  which given a mode  $m \in M$  lists the acceptable actions from  $\text{Act}$ , and the current mode *mode*, it holds

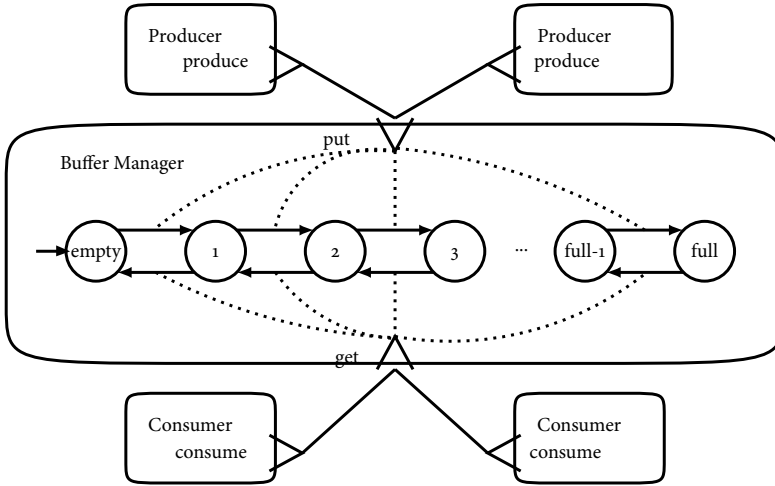


FIGURE 7.7 – Structure of the buffer management pattern.

that:

$$\forall a \in Act. \square(a \rightarrow a \in Acc(mode))$$

7.3.6 BUFFER MANAGEMENT

The *buffer manager pattern* can be used to manage producer-consumer systems with a shared, fixed-size buffer. A single managing component coordinates any number of producer and consumer operand components. Production and consumption of items is handled via the produce and consume input event ports, which block when the buffer is full respectively empty.

Given a buffer with capacity  $c$ , for each available buffer position  $n \in [0..c]$ , the manager specifies a mode  $mode_n$ , with transitions in between that synchronize on the produce and consume events, adding and removing an item from the buffer respectively. Each operand producer component can only add items to the buffer as long as it is not full, and similarly each consumer can only retrieve items as long as the buffer is not empty.

The pattern guarantees that items can only be retrieved as long as the buffer is not empty, and added as long as it is not full. Formally,

$$\begin{aligned} &\square(mode = mode_0 \rightarrow \neg get \mathcal{W} out) \wedge \\ &\square(mode = mode_n \rightarrow \neg put \mathcal{W} get) \end{aligned}$$

7.3.7 EVENT MONITORING

The *event monitoring pattern* permits an observer to monitor the occurrences of a particular event, and send one in response afterward. It is structurally equivalent

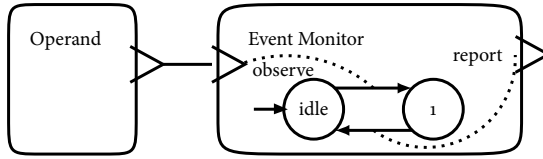


FIGURE 7.8 – Structure of the event monitor pattern.

to the buffer management patterns with a buffer size of one, where the producer generates the events being observed, and the consumer provides the reporting of those events. This structure ensures that each occurrence of the event being monitored is being responded to before another one can occur.

No assumptions are placed on the operands, and the pattern guarantees that:

$$\begin{aligned} & \square(\text{observe} \rightarrow \diamond \text{report}) \wedge \\ & \square(\text{observe} \rightarrow \neg \text{observe} \mathcal{U} \text{report}) \wedge \\ & \neg \text{report} \mathcal{W} \text{observe} \wedge \\ & \square(\text{report} \rightarrow \neg \text{report} \mathcal{W} \text{report}) \end{aligned}$$

### 7.3.8 PRIORITY MANAGEMENT

The *priority management pattern* allows specific operations to be performed according to some predefined strict priority order. The operation with the highest priority that is enabled being selected.

The operator component, i.e., priority manager, controls the priorities, and determines the action to be taken by means of an output event port for each action, which operand components synchronize on.

The approach taken for SLIM deviates slightly from that of the pattern given for BIP. The BIP approach uses a `noAction` event to signal a component is not available. For SLIM, a data flow is used to indicate readiness, allowing the model to be simplified, as the combination of `action` and `noAction` is no longer required to be input enabled.

The priority manager has a mode for each operand component. In that mode, one outgoing transition synchronizes on the `action` event, triggering that component. Another transition triggers on the internal event  $\tau$ , but is guarded by the negation of the enabled input of its respective component. This transition leads to the mode associated with a component of lower priority.

At any given time, at least one client operand must be enabled. Otherwise, a direct loop is possible between all states of the priority manager. Such a loop may prevent any other mode from being reached if no fairness constraint prohibits this (i.e., the priority manager's internal transitions preempt any other). The formalized

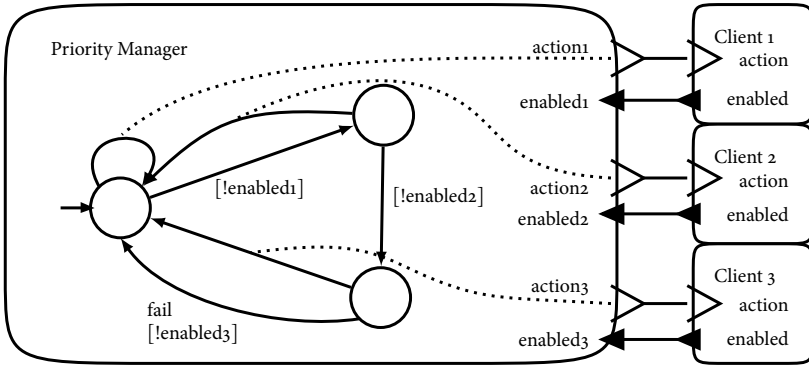


FIGURE 7.9 – Structure of the priority management pattern.

assumption is

$$\square(\bigvee_{i \in [1..n]} enabled_i)$$

The priority manager guarantees that if two operand components are enabled, the one with a higher priority will be triggered. Formally:

$$\square(\forall i \in [1..n]. action_i \rightarrow \bigwedge_{j \in [1..i-1]} \neg enabled_j)$$

## 7.4 MODEL

The case study has been modeled in SLIM based on the BIP model that is publicly accessible[101]. Where appropriate, the patterns described in the previous section have been used. The following will discuss the model in more detail, in particular with respect to the steps taken to translate it from BIP to SLIM.

The CubETH satellite is comprised of five systems (i.e., a system-of-systems), which are the electrical power system (EPS), the CDMS, the communication system (COM), the attitude determination and control system (ADCS) and the Payload (PL). In addition there is the mechanical structure which includes the antenna deployment subsystem. The case study models the software that runs on the CDMS system, which controls and monitors the various other subsystems that are running on-board the satellite. The CDMS monitors these subsystems, retrieves engineering data, and oversees the payload operations which are run as *scenarios*.

### 7.4.1 REQUIREMENTS

The design of the model is driven by a number of requirements specified for the CDMS system, provided in [101]. These requirements are listed in the following

table. Here,  $\varphi$  indicates whether a corresponding formal property exists, which will be further discussed in Section 7.5.1.

ID	Description	$\varphi$
CDMS-001	The CDMS shall be connected to the following subsystems: PL, COM, EPS through an I <sup>2</sup> C bus.	
CDMS-003	The CDMS shall supervise the correct execution of the software functions on the other subsystems. If a sensor or subsystem indicates anomalous signals the CDMS shall ask the EPS for a reset of the malfunctioning hardware.	
CDMS-004	The CDMS shall be able to save its status in order to resume correct operations following an unexpected reset.	
CDMS-006	The CDMS shall manage the data generated from the payload and housekeeping routines in a non volatile memory.	
CDMS-007	The CDMS shall periodically reset both the internal and external watchdogs and contact the EPS subsystem with a “heartbeat”.	✓
PL-001	The Payload shall be able to add a scenario to the payload board.	✓
PL-002	The Payload shall be able to execute scenario telecommand.	✓
PL-003	The Payload shall be able to abort any operation on the payload and data transfer to transfer data from the payload to the non-volatile memory.	✓
PL-004	The Payload shall be able to check the advancement of the payload board internals algorithms.	✓
PL-005	The Payload shall be able to track the upload, execution and result retrieval of a scenario and enable the corresponding actions.	✓
PL-006	The Payload subsystem shall have the following modes: IDLE, SCENARIO_READY, STARTED and RESULT_READY.	✓
PL-007	The payload shall operate when it is not in IDLE mode.	✓
PL-008	In SCENARIO_READY mode a scenario shall be loaded on the payload board.	✓
PL-009	In STARTED mode, the payload data acquisition shall begin.	✓
PL-010	The payload shall poll the payload board to check if its memory is full. If the memory is full, the payload shall change to RESULT_READY mode.	✓
PL-011	In RESULT_READY mode, the data shall be transferred to the CDMS non-volatile memory. If the data retrieval is not finished, payload shall continue the payload data acquisition until the data retrieval is completed.	✓
HK-001	The CDMS shall have a Housekeeping activity dedicated to each subsystem.	

HK-003	When line-of-sight communication is possible, housekeeping information shall be transmitted through the COM subsystem.	✓
HK-004	When line-of-sight communication is not possible, housekeeping information shall be written to the non-volatile flash memory.	✓
HK-005	A Housekeeping subsystem shall have the following states: NOMINAL, ANOMALY, and CRITICAL_FAILURE.	✓
HK-006	In NOMINAL state, the subsystem shall perform correctly.	✓
HK-007	If a process failure occurs or if the engineering data are not correct the subsystem shall enter the ANOMALY state.	✓
HK-008	After MAX seconds in ANOMALY, the subsystem shall enter the CRITICAL_FAILURE state.	✓! <sup>3</sup>
HK-009	In CRITICAL_FAILURE state, the subsystem shall contact the EPS and demand a restart of the malfunctioning subsystem.	✓
HK-010	During NOMINAL operation the subsystem shall be contacted to retrieve engineering data.	✓
I2C-001	A single user shall send one message at a time.	✓
I2C-002	I2C_sat shall implement the I <sup>2</sup> C protocol.	
Log-001	Every time a hardware error is produced, it shall be stored in a memory region in the RAM.	✓! <sup>4</sup>
Log-002	The dedicated RAM region shall be read and written atomically.	
Mem-001	The Central Software System shall have a dedicated Flash Memory Manager activity for managing flash memory operations.	
Mem-002	Flash memory shall be read and written atomically.	✓
Mem-003	Flash Memory Manager shall return the SUCCESS or FAIL status for each requested operation.	✓
Mem-004	Upon a read request, the Flash Memory Manager shall read the data from the flash memory and perform the cyclic redundancy check (CRC).	✓
Mem-005	If CRC fails, the Flash Memory Manager shall reread the data from the flash memory.	✓
Mem-006	For the same read request, the number of attempts by the Flash Memory Manager to read data from the flash memory shall have a value not larger than the parameter MAX_FM_READS <sup>5</sup> .	✓

<sup>3</sup>Not verified since the model is kept discrete and MAX is not specified. Instead it is asserted the state is reachable.

<sup>4</sup>Not verified as it is part of the complete model, see Section 7.5.2.

<sup>5</sup>As MAX\_FM\_READS itself is not specified, two is used as this allows for repetition with the smallest possible state space.

Mem-007	If the number of attempts by the Flash Memory Manager to read data from the flash memory exceeds MAX_FM_READS, the read operation shall be abandoned and a failure shall be reported.	✓
---------	---	---

### *Non-formalized Requirements*

Not all requirements have been formalized, either because they do not relate to (the behavior of) the system, or because they are too abstract in nature (cf. Section 3.1.1). The following lists these requirements and the reason they were not formalized.

- » CDMS-001: This requirement directly relates to the structure of the model. The connections referred to are present.
- » CDMS-003: The supervisory functionality is present by means of the housekeeping components and their failure monitoring aspect. However, the reset behavior described is not modeled, as it is part of the EPS system which is not modeled.
- » CDMS-004: This functionality is not present in the model.
- » HK-001: The requirement describes the existence of the housekeeping components for the PL, COM and EPS subsystems, which are present.
- » Mem-001: This requirement describes the existence of the flash memory subsystem, which is present in the model.

### *Requirements Discussion*

As discussed in Chapter 3, it is well known that requirement specification is non-trivial, and as a result the requirements may contain ambiguities or errors. Since the model is based on these requirements, care has to be taken that they are correct and reflect the intended outcome. However, for some of the requirements above a different interpretation is required, as is described below.

**Failure Management** Requirement HK-006 specifies that in the NOMINAL state a subsystem shall perform correctly. This is technically infeasible since the housekeeping component can only react to misbehaving subsystems, and thus at least briefly remains in the NOMINAL state until it switches to ANOMALY. Instead, it will be assumed the requirement means “In NOMINAL state, an error shall trigger a transition away from the NOMINAL state.”

Furthermore, no requirement specifies that a recovery may be possible when in the ANOMALY state. The case study does assume this, as it is a desired function (in order to recover from glitches more easily). Such a requirement should be present, as otherwise an implementation without this function is still valid.

**Flash Memory** The requirements for the flash memory read functionality specify that a failed read should be attempted again a number of times before giving up on the operation. However, they are slightly inconsistent: Mem-005 requires data be reread if an attempt fails, but does not specify an upper bound (nor is one implied), instead this is specified by Mem-006. Taken literally, this means requirement Mem-005 cannot be satisfied, as at some point a failed CRC will cause the read operation to stop according to Mem-006. Instead, Mem-005 is assumed to state “If CRC fails *and the number of read attempts is below MAX\_FM\_READS*, the Flash Memory Manager shall reread the data from the flash memory.”

Requirement Mem-007 specifies that the operation fails if the number of read operations exceeds MAX\_FM\_READS, however, Mem-006 prohibits this. Instead, Mem-007 is assumed to state “If the number of attempts by the Flash Memory Manager to read data from the flash memory *is equal to MAX\_FM\_READS*, the read operation shall be abandoned and a failure shall be reported.”

#### 7.4.2 COMPONENTS

For each major subsystem a description will be given of its function, as well as how it is modeled. Note that reused components are not listed individually, as they can be trivially reused as subcomponents.

##### *Mutex Component*

A Mutex component is defined of which instances are used throughout the model to prevent concurrent access to single resources. The mutex is implemented as a simple two-state state machine, the initial state allowing an event to enter the critical section of the mutex, and the second state allowing an event to leave. Functionally and structurally, it is the same as the model shown for the Mutual Exclusion pattern show in Section 7.3.1.

##### *I<sup>2</sup>C*

Most of the high-level communications across the satellite occur via an I<sup>2</sup>C interface. The protocol used by the CDMS software first sends a request packet via I<sup>2</sup>C to the target component (which in terms of I<sup>2</sup>C is a slave, the CDMS component being the master). After this, the I<sup>2</sup>C bus is polled for a response by sending a poll message until a response is received, or a timeout occurs.

In the case study, the I<sup>2</sup>C messages are abstracted by the MessageLibrary component, which provides functions, modeled as events, that allow encoding a request, and decoding the response (or error) data. The MessageLibrary then interfaces with the I2C\_sat component. This component provides the read and write functions, again modeled as events, which implement the protocol used to send and receive messages via the I<sup>2</sup>C bus. See Figure 7.10 for the model of the MessageLibrary component. When encoding a request, an I<sup>2</sup>C write request is sent, and it then awaits a response.

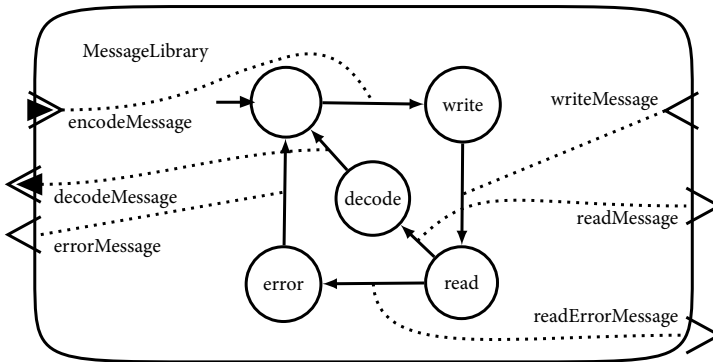


FIGURE 7.10 – Model of the MessageLibrary subsystem.

The `encodeMessage` port is an event data port that accepts an integral number representing the address of the subsystem the message is directed to. The result is returned via the Boolean event data port `decodeMessage`, the value of which is determined non-deterministically. The meaning of the result depends on the receiving component. Failures are indicated with the event port `errorMessage`.

The model of the `I2C_sat` subsystem is based on a single component `I2C_flow`, which mirrors the BIP model from [115], and the `I2C_satLibrary` library component. The complete model is shown in Figure 7.11.

The behavior is modeled by a single state machine that follows the protocol, performing the request `write`, followed by a `write-read` operation polling for the response. The model inspects the CRC of the response data, and retries the operation if it is incorrect, and otherwise returns the data. It is possible for the request `write` operation to fail, and the `read` operation to fail. A number of attempts is possible (modeled non-deterministically) before a failure is returned.

Note that this model does not follow that of [102], but rather its predecessor. This is because of a lower complexity, and to allow for the `write-read` behavior originally designed for.

The `I2C_satLibrary` component provides an interface to the `I2C` hardware made available via two commands, represented by event ports: `masterWrite` and `masterRead`. The first will send some message over the `I2C` bus, the second will check if some data is available on the bus. A third event data port `check_CRC` models the CRC function, and chooses its value non-deterministically.

The `MessageLibrary` and `I2C_sat` components are wrapped in a `I2Cbus` component to ease the use of them in the case study. This is, however, only a thin wrapper around the provided port connections. The complete system contains one instance of the `I2Cbus` component. Due to this, it has to be ensured that the `decodeMessage` and `errorMessage` ports are connected to input-enabled (or nonblocking) input



- » A component `PacketStoreModeManager` which controls whether data is sent to flash memory or sent via telemetry, tracking and control (TTC) (see Section 7.4.2), implemented using the mode-manager pattern;
- » A component `FailureMonitoring` which monitors for any critical failures during housekeeping operations, and reports them as necessary, making use of the failure-monitoring pattern, and
- » A component `HKProcess` which models the steps required to process the engineering data using the action-flow pattern.

For the CDMS housekeeping subsystem, the data collection and failure monitoring are handled differently, so a different component implementation exists. It shares the same mutex, `ModeManager` and `PacketStoreModeManager` components, but has its own `HKCDMSProcess` component for its behavior.

The shared components are depicted in Figure 7.12. The mode manager component enabled or disables the housekeeping activities by selectively allowing the main `read_HK` event. The packet store mode manager determines how retrieved data is stored, by selectively allowing the `mem_write_req` or `i2c_ask_ttc` events. The former writes data to non-volatile memory, the latter to the COM system. The failure monitoring component sends a signal to the EPS subsystem in case of a failure, which on the platform will cause the EPS subsystem to perform a reset of the corresponding component (as these components are not modeled, neither is this process). This communication is also performed via I<sup>2</sup>C.

The main loop of operations of the housekeeping component, shown in Figure 7.13, is periodically executed (triggered by the `read_HK` event). At the start, engineering data is retrieved via I<sup>2</sup>C, which then gets stored either in non-volatile memory or sent to the communications subsystem via I<sup>2</sup>C. The choice on which to perform depends on the presence of a line-of-sight connection to ground, discussed below. A error indicated via I<sup>2</sup>C triggers an `anomaly` event, which is forwarded to the failure monitoring component.

The housekeeping component of the CDMS is functionally nearly identical to those for the EPS, PL and COM systems. However, the engineering data is retrieved via internal means, such as general purpose IO (IO) and state registers, as opposed to using I<sup>2</sup>C. Furthermore, the status of the CDMS subsystem is monitored separately, so no failure monitor is included. Its model is shown in Figure 7.14.

**Housekeeping Control Components** The housekeeping components can be enabled or disabled, as well as switch between storing data or sending it to ground, by means of certain telecommands. The housekeeping control commands are handled by the `s3_5` and `s3_6` components (shown in Figure 7.15), which handle the `enable_HK` and `disable_HK` commands respectively. Upon reception, they send an event to all housekeeping components to switch the mode of the `ModeManager`.

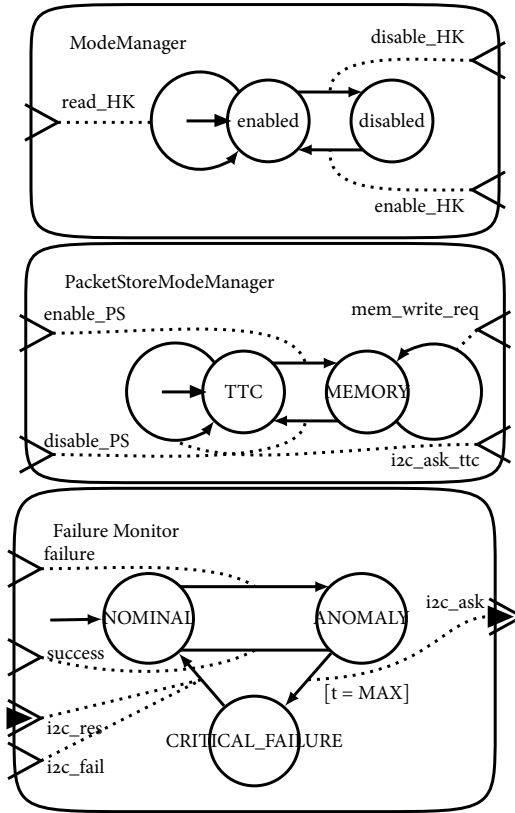


FIGURE 7.12 – Shared housekeeping components.

The data handling mode is controlled by the `s15_1` and `s15_2` components (see Figure 7.16), handling the `enable_PS` and `disable_PS` commands respectively. These are forwarded to the `PacketStoreModeManager` components.

Different between the BIP and SLIM models is that for BIP, multiple event synchronizations occur for each housekeeping component, with a priority to enforce the switch. In SLIM, a fan-out event connection is used to control them simultaneously.

#### *Command and Data Management Subsystem Monitoring*

The status of the CDMS is monitored by the `CDMSstatus` subsystem. It checks, with the help of a watchdog, that the CDMS is functioning correctly.

The `CDMSstatus` component, cf. Figure 7.17, consists of a watchdog component and an action-flow component. The watchdog component is responsible for resetting both the internal and external watchdog timers of the actual implementation. This is done periodically, as the expiration of a timer indicates a problem with the CDMS.

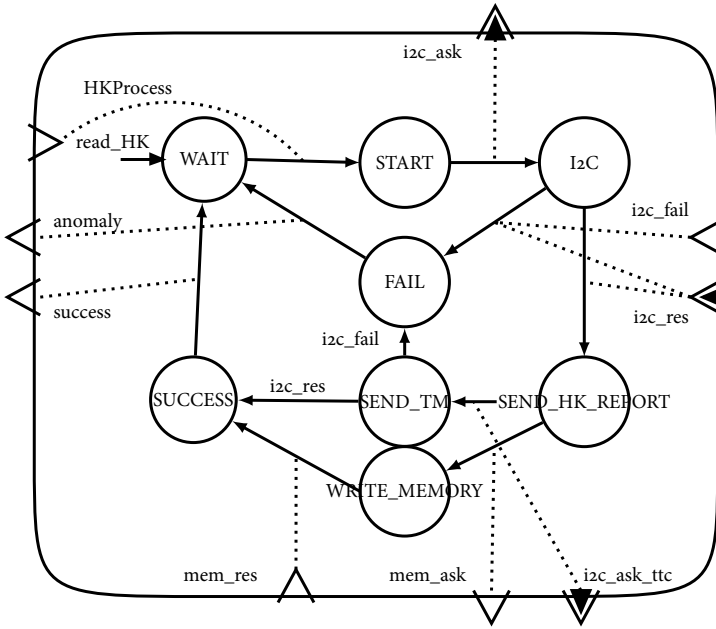


FIGURE 7.13 – Model of the housekeeping process for EPS, PL and COM.

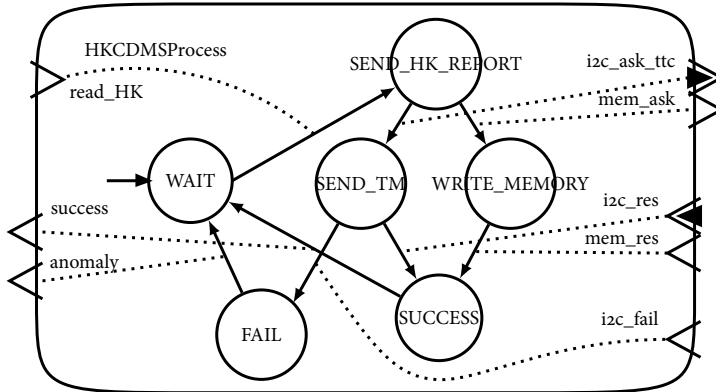


FIGURE 7.14 – Process of the CDMS housekeeping subsystem.

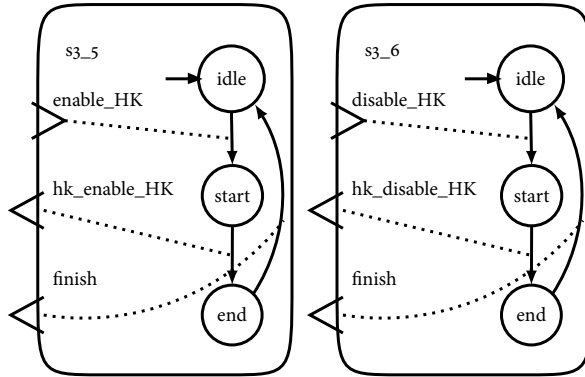


FIGURE 7.15 – Housekeeping control command model.

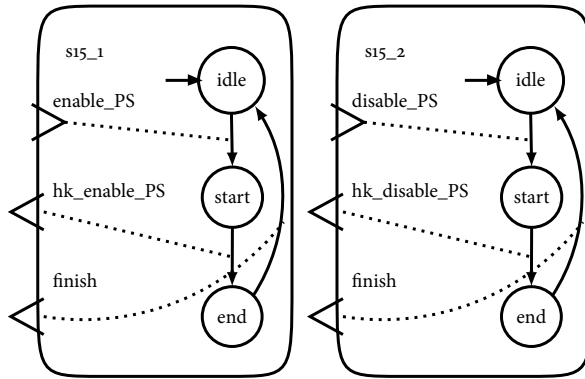


FIGURE 7.16 – Housekeeping data handling command model.

In the model, the time period is abstracted to a single event, which may non-deterministically be triggered. The action-flow component controls the order of events. First the internal and external watchdog timers are reset (in that order), followed by sending a heartbeat (via I<sup>2</sup>C) to indicate the system is operational. Finally, it awaits the result of sending the heartbeat, and the loop repeats.

### *Payload and Scenarios*

The payload operations of the satellite are controlled by the Payload component, which itself is composed of various subsystems. These systems control the execution of (mission) scenarios, data storage and monitoring.

The payload component can operate in four modes, controlled by a mode-manager subcomponent (PayloadModeManager, see Figure 7.18). These are respectively IDLE, SCENARIO\_READY, STARTED and RESULT\_READY. In these modes respectively

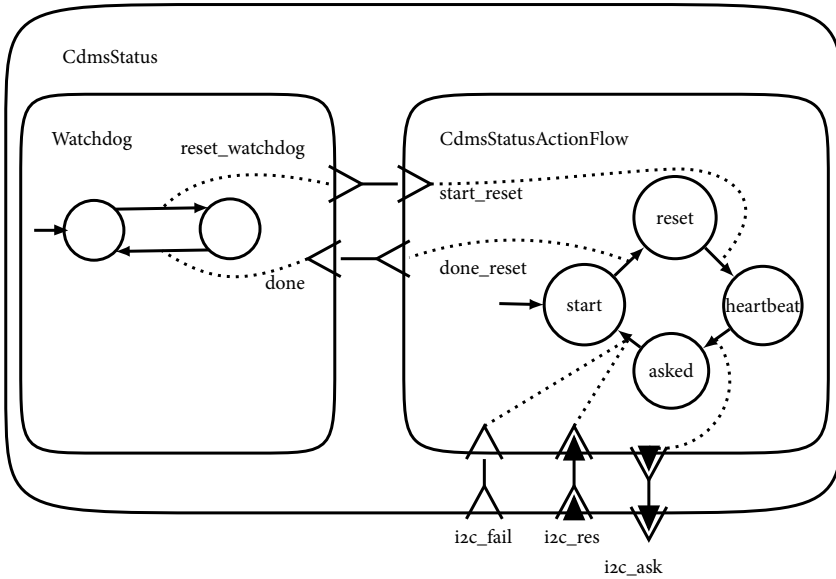


FIGURE 7.17 – Model of the CDMS status subsystem.

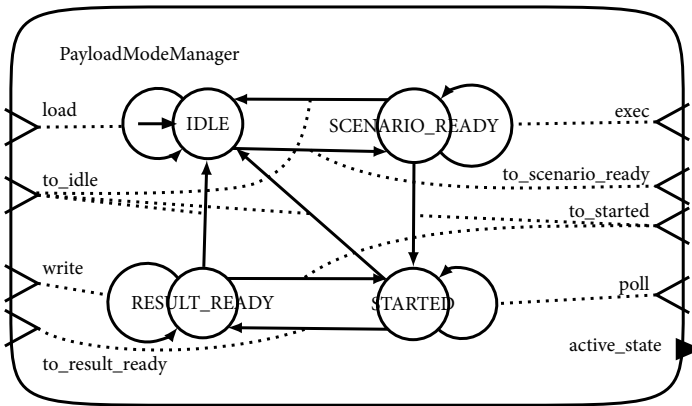


FIGURE 7.18 – The payload mode manager component.

a scenario may be loaded, is loaded, is executing and finally is ready for collecting result data.

The loading and execution of the scenarios is handled by a different set of components. The s128\_1, s128\_4 and s128\_5 components are responsible for handling the possible telecommands (LOAD, EXEC and ABORT respectively). The s128\_1 component is responsible for loading a new scenario onto the payload, and s128\_4 for its execution. The s128\_5 component permits the scenario to be aborted, resetting

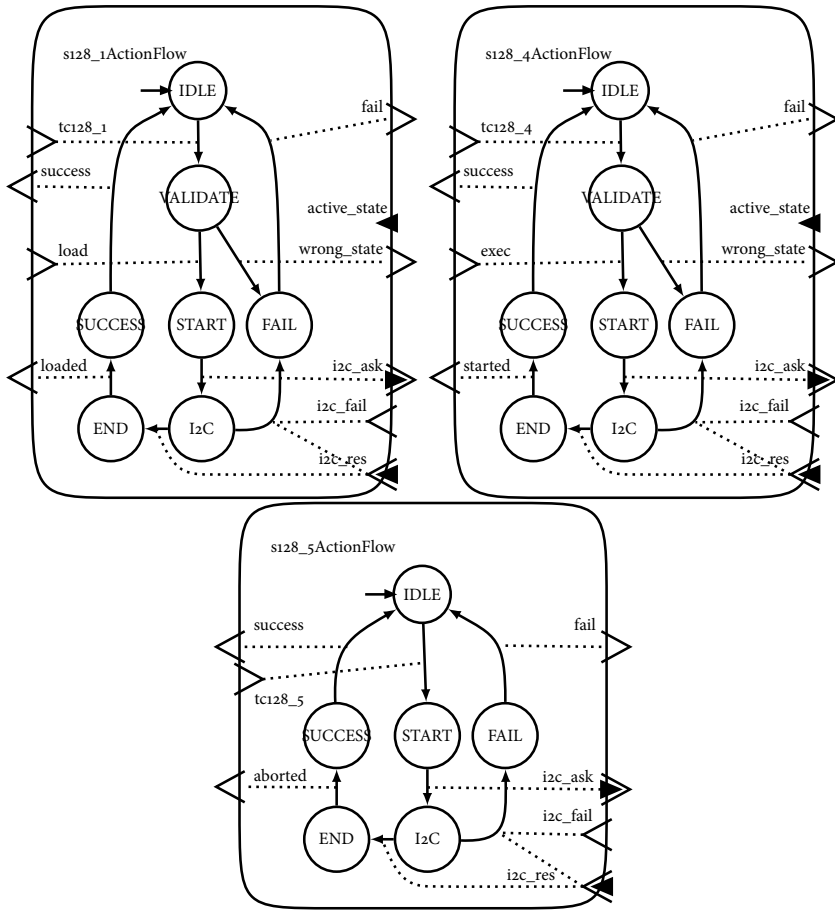


FIGURE 7.19 – Models of the telecommand processing subsystems.

the payload manager to the IDLE mode. Their models can be found in Figure 7.19.

The `s128_1` and `s128_4` components are similar to each other, except that they operate in different modes, IDLE and SCENARIO\_READY respectively. After receiving the telecommand, it checks if the payload is in the correct mode. If so, the command is sent to the payload via I<sup>2</sup>C, and on success the mode is switched accordingly. If the mode is invalid or the I<sup>2</sup>C operation failed, the component returns to the initial state and sends a failed event.

If a scenario has been loaded and is executing, the `status_verification` component continuously polls the status of the scenario via I<sup>2</sup>C. When the scenario has finished, it will trigger a transition back to the IDLE mode. If all available memory has been filled in the payload, it will initiate a transfer of the collected data by switching the payload to the RESULT\_READY mode. The model is shown in

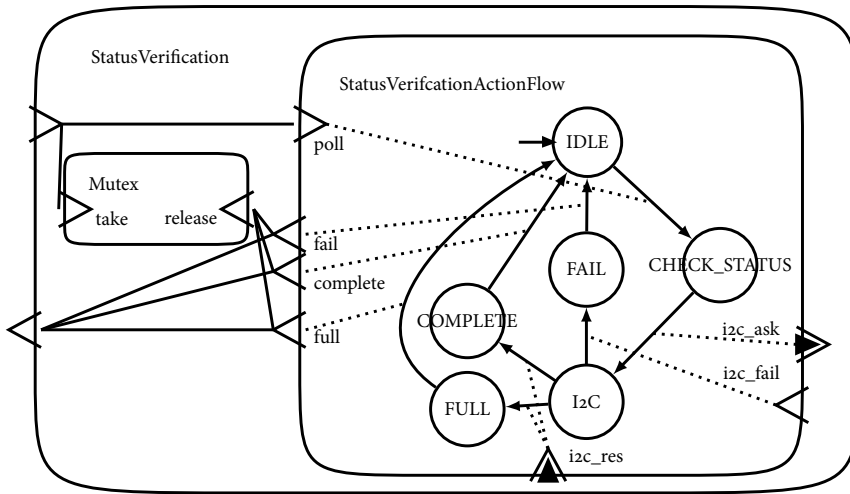


FIGURE 7.20 – Model of the status verification subsystem.

Figure 7.20.

The data transfer itself is managed by the `data_transfer` component, shown in Figure 7.21, which moves the data to the non-volatile memory when the payload is in the `RESULT_READY` mode. It consists of a mode manager to control when data write operations should be performed, and an action flow to do so. When writing, i.e., the mode manager is in the `BUSY` mode, the action flow component requests the data from the payload via I<sup>2</sup>C, and writes it to non-volatile memory via the `FlashMemory` interface (see Section 7.4.2). The payload I<sup>2</sup>C result indicates whether there is more data to be written or not. If not, the operation finished and the payload mode manager is instructed to switch back to the `STARTED` mode.

At any point during the loading or executing of a scenario, it may be aborted. This process is handled by the `128_5` component, which performs the operation after receiving the corresponding telecommand. Functionally it is similar to the `128_1` and `128_4` components, but does not validate the current payload mode, since the abort command can always be sent.

A difference between the BIP and SLIM models is the handling of telecommands that are sent out of order. Such commands trigger the `wrong_state` event. For instance when the `LOAD` command is sent if the payload is in the `SCENARIO_READY` mode. In the BIP model, priorities are used to trigger this transition only when the alternate transition (e.g., `load` or `exec`) is not enabled because it cannot synchronize with the mode manager. In SLIM this is not possible, as there is no support for priorities. Instead, the mode manager has a data port of which the value depends on the active mode. This value is checked, and when incorrect allows the `wrong_state` event to be triggered.

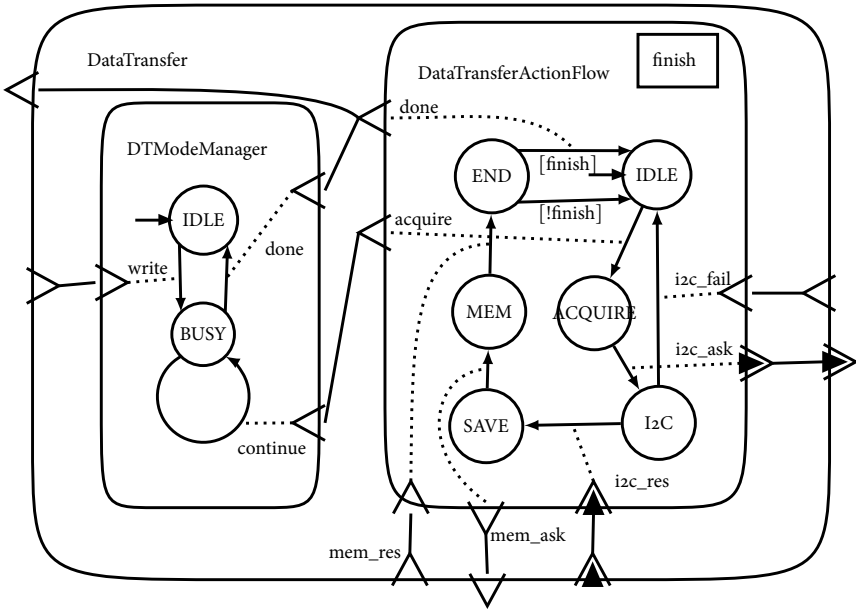


FIGURE 7.21 – Model of the data transfer subsystem.

*Storage*

The data that is collected during operation of the satellite is considered in terms of packets, and can be handled in either of two ways: Sending it to the ground station directly, or storing it in non-volatile (NOR flash) memory. The operation of the latter is represented by the FlashMemory component, and a library component MemoryLibrary.

The FlashMemory component, shown in Figure 7.22, controls the read and write actions, and is responsible for verifying them. It consists of various subcomponents to manage the operation of the memory. To ensure the memory is accessed by one system at a time, a standard Mutex component is used, which needs to be acquired to perform a read or write operation. Two mode managers handle the read and write states of the memory, ReadModeManager and WriteModeManager respectively. Both mode managers allow a read/write operation to be started, continued,



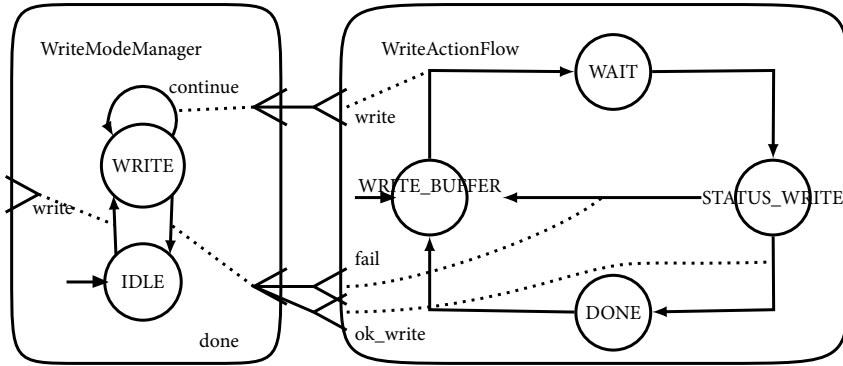


FIGURE 7.23 – Behavior of the flash write functionality.

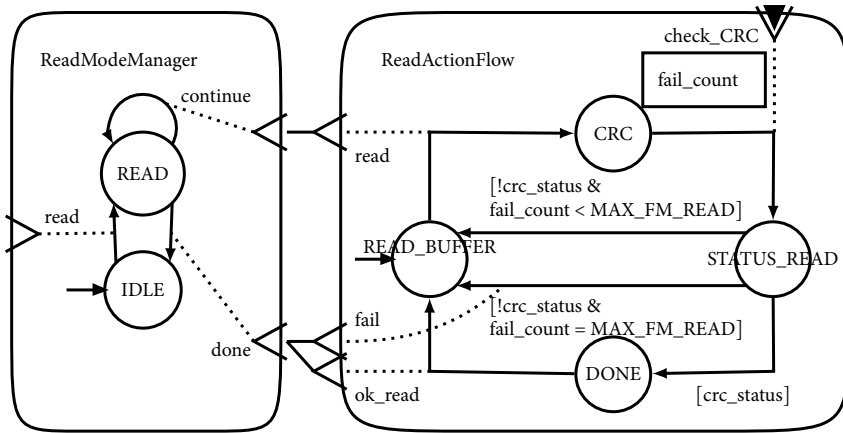


FIGURE 7.24 – Behavior of the flash read functionality.

subcomponent `fail_count`. If the CRC fails and the failure limit has not been reached, another read attempt is made. Otherwise, the operation fails.

The interface with the hardware is modeled by the `MemoryLibrary` component, which models the read and write mode, and CRC functions as events. Similar to `I2Csat_Library`, these are opaque event (data) ports.

### Logging

A separate region in RAM is used for storing logging data. A logging component `Logger` provides the interface to write logging information to it, in particular for (hardware) error events. It consists of a standard mutex to ensure one component can perform a logging operation at a time, and another, `Log`, to perform the actual operation. One abstract `log` event port is provided to trigger a log operation.

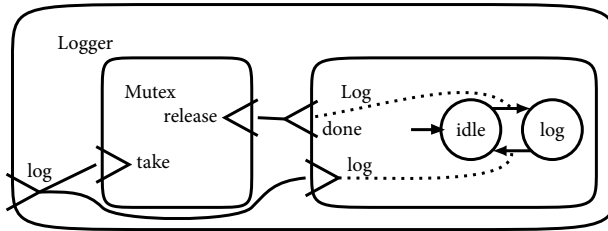


FIGURE 7.25 – Model of the logging subsystem.

### 7.4.3 DIFFERENCES BETWEEN SLIM AND BIP

The primary difference between the BIP and SLIM models of the CubETH satellite is the way event connections are structured and handled. Whereas the BIP model specifies multiple possible interactions using multiple port connectors, this is not possible in SLIM. Instead, all port connections are considered for the synchronization. As they must all synchronize for an event to be possible, shared events, such as those from the I<sup>2</sup>C bus are marked as nonblocking in leaf components.

This leads to some structural changes: The housekeeping components do not synchronize their mutexes with I<sup>2</sup>C events, but rather internal start and end events. This permits the I<sup>2</sup>C events to share a common event connection, as otherwise all mutexes would have to synchronize, which contradicts their purpose. For the telecommand events of the payload component similar restrictions apply. However, as their behavior is coordinated with a single mode manager, the mutexes have been removed, as they are redundant (since the individual payload components only accept telecommand events in their initial state).

The I<sup>2</sup>C messages are all encoded with an integer to identify them. As the I<sup>2</sup>C bus is a single mutual exclusive resource, this does not alter the reachable state space in the model, as each result event is preceded by a request event. However, it does open the possibility to do so, and makes inspecting traces of the model easier, since the origin of the messages can be traced.

To keep the size of the model bounded, which permits BDD-based model checking as discussed in Section 5.3.3, fixed-width integers are used (specified using range types). As the values of data components and ports are always guaranteed to remain within some fixed upper-bound, this does not affect the actual reachable states.

To simplify composition, the complete model contains the subsystems in a nested fashion. This makes it possible to perform analysis on parts of the model by specifying a different root components, without having to alter the model itself.

Finally, as SLIM does not impose fairness constraints on the model by default, it is possible for starvation to occur for parts of the system if a single component exhibits cyclic behavior. In the model, such cycles can be found in the (non-de-

terministic) read and write loops of both the I<sup>2</sup>C bus and non-volatile memory. Fairness constraints have been added explicitly to exclude such behavior, see below.

#### 7.4.4 FAIRNESS

As mentioned in Chapter 5, SLIM does not automatically impose any fairness restrictions on the possible state transitions. This means that when two components are competing, usually due to non-determinism, it is possible for one to starve the other.

In this case study, multiple sources of starvation exist. These include the occurrence of telecommands (which are sent from the environment, and are thus not controlled), various read/write operations of a non-deterministic length or non-deterministic number of possible failures, the CDMS watchdog, and the housekeeping data retrieval operations that run on a non-deterministic interval.

This means that, although each component can easily be verified individually, the whole system needs some amendments to limit the possible behavior to that which can be considered fair/realistic. Two approaches are possible: A wrapper component can be specified that controls the occurrence and order of external (and possibly internal) events, or fairness constraints can be added to the model to limit the verification to *realistic* traces.

The case study specifies a number of fairness constraints on the components for which it is known that they will always perform an operation at some point in the future. These operations include the housekeeping data retrieval, the payload scenario completion, the flash memory read/write operations and the CDMS watchdog reset. For the events associated with these operations, fairness constraints have been added.

One caveat should be noted: Similar to deadlocks, fairness constraints can have negative effects on the outcome of verification if they are incorrect. If a fairness constraint is too strong, it will limit the number of paths that are considered for model checking, making it impossible to find a counter-example of a property that realistically should not hold.

Therefore, care has to be taken that the number of such constraints is kept to a minimum, and that they are applied to events that both can and should occur infinitely often on paths of infinite length.<sup>6</sup>

## 7.5 ANALYSIS

Analysis of the model has been performed based on properties that are derived from the architectural patterns as presented in the BIP case study, translated to their SLIM equivalents. In addition, formal properties have been defined and analyzed

---

<sup>6</sup>A simple trick to validate the model is to add a trivially falsified property. If no counter-example is found, the model may not be correctly specified.

based on the requirement specifications provided in Section 7.4.1. However, those only referring to the structure of the system were not formalized. The table provided in Section 7.4.1 indicates for each requirement if it was formalized.

### 7.5.1 PROPERTIES

Properties are defined on the components to which they apply, which permits them to be verified for each individual instance, as well as in a compositional manner. In particular the mutex and housekeeping components are reused. Each instance of these properties is not individually listed, rather only the specification they are derived from.

In order to specify the properties in SLIM, the operators used by the pattern properties need to be mapped to SLIM specifications. The operator  $\square$  maps to **always**,  $\diamond$  to **in the future**,  $\circ$  to **then** and  $\mathcal{U}$  to **until**. For  $\varphi \mathcal{W} \psi$ , the SLIM equivalent is **then**( $\psi$ ) **releases**  $\varphi$ . Further details about these operators can be found in section A.6. For brevity, the prefixes for mode identifiers have been omitted (in SLIM, this is either the name of the subcomponent that defines them, or **mode**:).

Properties derived from the mode management architecture are labeled with “MD”, those from the mutual exclusion architecture “MX”, from the failure monitor architecture “FM” and action flow architecture with “AF”. Properties based on the requirements specified in Section 7.4.1 use the respective requirement ID.

**CDMS Status** For the CDMS status subsystem, some properties were defined to validate the action flow behavior. It checks that 1) the last action is not started before the first has been taken, 2) an operation does not repeat until the loop is taken again, and 3) the loop is not completed until the last action is taken. Additionally, requirement CDMS-007 was formalized using this component. These properties are:

ID	Property
CDMS.AF.1	$\square(\text{controller.mode} = \text{START} \rightarrow \neg \text{controller.i2c\_ask} \mathcal{U} \text{controller.start\_reset})$
CDMS.AF.2i	$\square(\text{controller.start\_reset} \rightarrow \circ(\neg \text{controller.start\_reset} \mathcal{W} \text{controller.mode} = \text{START}))$
CDMS.AF.2ii	$\square(\text{controller.i2c\_ask} \rightarrow \circ(\neg \text{controller.i2c\_ask} \mathcal{W} \text{controller.mode} = \text{START}))$
CDMS.AF.3	$\square(\text{controller.mode} = \text{START} \wedge \text{controller.start\_reset} \rightarrow \circ(\text{controller.mode} \neq \text{START} \mathcal{U} \text{controller.i2c\_res} \vee \text{controller.i2c\_fail}))$
CDMS-007	$\square(\diamond \text{watchdog.reset\_watchdog} \wedge \diamond \text{controller.i2c\_ask})$

**FlashMemory** For the FlashMemory and related components both properties for the mode management as well as the associated requirements were specified:

ID	Property
Mem.MD.1i	$\square(\text{read} \rightarrow \text{rmm.mode} = \text{READ})$
Mem.MD.1ii	$\square(\text{write} \rightarrow \text{wmm.mode} = \text{WRITE})$
Mem-002	$\square(\text{read} \rightarrow \circ(\neg \text{read} \mathcal{U} (\text{return} \vee \text{fail}))) \wedge$ $\square(\text{write} \rightarrow \circ(\neg \text{write} \mathcal{U} (\text{return} \vee \text{fail})))$
Mem-003	$\square(\text{read} \vee \text{write} \rightarrow \diamond(\text{return} \vee \text{fail}))$
Mem-004	$\square(\text{read} \rightarrow \diamond \text{rafwa.read} \wedge \diamond \text{rafwa.check\_CRC})$
Mem-005	$\square(\text{rafwa.check\_CRC} \wedge \neg \text{data}(\text{rafwa.check\_CRC}) \wedge$ $\text{rafwa.fail\_count} < \text{MAX\_FM\_READ} \rightarrow$ $\neg(\text{rafwa.fail} \vee \text{rafwa.ok\_read}) \mathcal{W} \circ \text{rafwa.read})$
Mem-006	$\square(\text{rafwa.fail\_count} \leq \text{MAX\_FM\_READ})$
Mem-007	$\square(\text{rafwa.check\_CRC} \wedge \neg \text{data}(\text{rafwa.check\_CRC}) \wedge$ $\text{rafwa.fail\_count} = \text{MAX\_FM\_READ} \rightarrow$ $(\text{rafwa.read} \vee \text{rafwa.ok\_read}) \mathcal{W} \circ \text{rafwa.fail})$

The original property MX.1 has not been translated as it was too strong: It specified an invariant that held as long as not all components were waiting for the flash memory component. It, however, did not consider all subsets of components. Instead, the translated model relies on the SLIM semantics, as the `read` and `write` events are inputs, meaning they can synchronize with only one component at a time. Requirement Mem-002 ensures mutual exclusion of all operations.

**HouseKeeping** The housekeeping components had mode management and failure management properties verified. As there were multiple instances of the housekeeping components (EPS, PL and COM), multiple instances of these properties were verified as well. In addition, the requirements directly related to these components were formally specified.

ID	Property
PS.MD.1i	$\square(\text{packet\_store\_MM.mem\_write\_req} \rightarrow$ $\text{packet\_store\_MM.mode} = \text{MEMORY})$
PS.MD.1ii	$\square(\text{packet\_store\_MM.ask\_i2c\_ttc} \rightarrow$ $\text{packet\_store\_MM.mode} = \text{TTC})$
HK.MD.1	$\square(\text{process\_AF.read\_HK} \rightarrow \text{enable\_MM.mode} = \text{ENABLED})$
HK.FM.1	$\square(\text{process\_AF.anomaly} \rightarrow$ $\circ(\neg \text{process\_AF.anomaly} \mathcal{U}$ $(\text{process\_AF.success} \vee \text{fail\_mon.i2c\_ask})))$
HK-003	$\square(\text{enable\_PS} \rightarrow$ $\neg \text{packet\_store\_MM.ask\_i2c\_ttc} \mathcal{W} \text{disable\_PS})$ $\square(\text{disable\_PS} \rightarrow$ $\neg \text{packet\_store\_MM.mem\_write\_req} \mathcal{W} \text{enable\_PS})$

HK-004	$\square(\text{enable\_PS} \rightarrow$ $\quad \circ(\text{packet\_store\_MM.mode} = \text{MEMORY} \mathcal{W} \circ \text{disable\_PS})) \wedge$ $\square(\text{packet\_store\_MM.mode} = \text{MEMORY} \rightarrow$ $\quad \neg \text{packet\_store\_MM.i2c\_ask\_ttc})$
HK-005	$\square(\text{fail\_mon.mode} = \text{NOMINAL} \vee \text{fail\_mon.mode} = \text{ANOMALY}$ $\quad \vee \text{fail\_mon.mode} = \text{CRITICAL\_FAILURE})$
HK-006	$\square(\text{fail\_mon.mode} = \text{NOMINAL} \wedge \text{fail\_mon.failure} \rightarrow$ $\quad \circ(\text{fail\_mon.mode} \neq \text{NOMINAL}))$
HK-007	$\square(\text{fail\_mon.mode} = \text{NOMINAL} \wedge \text{fail\_mon.failure} \rightarrow$ $\quad \circ(\text{fail\_mon.mode} = \text{ANOMALY}))$
HK-008	$\square(\text{fail\_mon.failure} \rightarrow$ $\quad \text{time\_until}(\text{fail\_mon.mode} = \text{CRITICAL\_FAILURE}) = \text{MAX}$ $\quad \vee \text{time\_until}(\text{fail\_mon.success}) < \text{MAX})$
HK-009	$\square(\text{fail\_mon.mode} = \text{CRITICAL\_FAILURE} \rightarrow$ $\quad \diamond \text{fail\_mon.i2c\_ask})$
HK-010	$\square(\text{process\_AF.read\_HK} \rightarrow \text{fail\_mon.mode} = \text{NOMINAL})$

The failure monitoring property was altered to check that after an anomaly has occurred, either a recovery happens or an error report is sent before another anomaly occurs. Originally, it checked that the housekeeping component did not finish before a recovery event or an error report message.

The mutual exclusion properties (MX.1) were not considered, the reason being that the periodic `read_HK` event is not externally accessible. Between the various housekeeping components only the I<sup>2</sup>C bus is shared, which itself guarantees mutual exclusion.

**HouseKeeping CDMS** For the CDMS housekeeping component, the same properties were specified as those for the other housekeeping components, with the exception of HK.FM.1 and HK-005–HK-009, because failure monitoring is not included.

**I<sup>2</sup>C** For the I2C\_sat component, a single mode management property was specified in [102]. Requirement I2C-001 was specified against the MessageLibray component. The mode management property is only applicable to the component implementation making use of the mode management pattern, and not the component implementation using a single flow (from [115]), which is the one adopted in this case study. A translated version is provided, though it is not verified.

ID	Property
MD.1	$\square(\text{reader.poll} \rightarrow \text{controller.mode} = \text{SI})$
I2C-001	$\square(\text{encodeMessage} \rightarrow$ $\quad \circ(\neg \text{encodeMessage} \mathcal{U} (\text{decodeMessage} \vee \text{errorMessage})))$

**Payload** The original case study defined properties both for validating the mutual exclusion pattern and the mode management pattern. Of these, only the mode management patterns were considered, with in addition properties for some of the payload requirements, as listed below:

ID	Property
MD.1i	$\square(s128\_4.exec \rightarrow mgr.mode = SCENARIO\_READY)$
MD.1ii	$\square(sv.started \rightarrow mgr.mode = STARTED)$
MD.1iii	$\square(s128\_4.executed \rightarrow mgr.mode = IDLE)$
MD.1iv	$\square(dt.ready \rightarrow mgr.mode = RESULT\_READY)$
DT.MD.1	$\square(dt.proc.acquire \rightarrow dt.manager.mode = BUSY)$
PL-001	$\square(tc\_ask \wedge data(tc\_ask) = LOAD \rightarrow \diamond s128\_1.load)$
PL-002	$\square(tc\_ask \wedge data(tc\_ask) = EXEC \rightarrow \diamond s128\_4.exec)$
PL-003	$\square(tc\_ask \wedge data(tc\_ask) = ABORT \rightarrow \diamond(s128\_5.aborted \vee s128\_5.fail))$
PL-004	$\square(mgr.mode = STARTED \rightarrow \diamond(sv.poll \vee s128\_5.aborted))$
PL-005	$\square((mgr.load \rightarrow mgr.mode = IDLE) \wedge (mgr.exec \rightarrow mgr.mode = SCENARIO\_READY) \wedge (mgr.poll \rightarrow mgr.mode = STARTED))$
PL-006	$\square(mgr.mode = IDLE \vee mgr.mode = SCENARIO\_READY \vee mgr.mode = STARTED \vee mgr.mode = RESULT\_READY)$
PL-007	$\square(mgr.mode = IDLE \rightarrow (mgr.mode = IDLE \mathcal{W} s128\_1.loaded)) \wedge \square(mgr.mode \neq IDLE \rightarrow (mgr.mode \neq IDLE \mathcal{W} (sv.complete \vee s128\_5.aborted)))$
PL-008	$\square(mgr.mode = SCENARIO\_READY \rightarrow \diamond(s128\_4.started \vee s128\_5.aborted))$
PL-009	$\square(mgr.mode = STARTED \rightarrow \diamond(sv.full \vee sv.complete \vee s128\_5.aborted))$
PL-010	$\square(mgr.mode = STARTED \wedge sv.full \rightarrow \diamond(mgr.mode = RESULT\_READY))$
PL-011	$\square(mgr.mode = RESULT\_READY \rightarrow \diamond(dt.manager.continue \mathcal{U} (dt.manager.done \vee s128\_5.aborted)))$

The mutual exclusion properties were not considered, as they assume the existence of multiple instances of the payload components. Instead, mutual exclusion is guaranteed by the fact that there is a single input event (data) port for the telecommands that control the s128 components.

**Logging** The logging component uses a mutex component to ensure only one log operation can be performed at the same time. The original case study defined a property to validate this. However, similar to the the flash memory component, it

TABLE 7.10 – Overview of verified components and the number of properties analyzed.

Component	Components	States	Properties (inherited)
CubETH	44	$2.73 \cdot 10^{11}$	-
Payload	9	355 290	15 (+2)
I2CBus	5	9 728	1
HK	6	184	10 (+1)
HKCDMS	5	88	3 (+1)
FlashMemory	7	40	10 (+1)
CDMSStatus	3	14	5
Logger	3	2	0 (+2)
Mutex	1	2	2

is specified too strong. Instead, it is checked that the logger is available only when idle, and is not idle when a logging operation occurs. In addition, requirement Log-001 was formalized against the complete model:

ID	Property
MX.1	$\square(\log \rightarrow \circ(\neg \log \mathcal{U} \text{loglib.mode} = \text{IDLE}))$
AF.1	$\square(\log \rightarrow \circ(\text{loglib.mode} = \text{LOG}))$
Log-001	$\square(\text{flash\_mem.fail} \vee \text{i2c\_bus.errorMessage} \vee \text{payload.tc\_fail}) \rightarrow \text{log.log}$

**Mutex** The following properties were defined for the mutex component. Effectively, they check that a takes mutex cannot be taken until released, but in addition check that the mutex is always eventually taken. The latter ensures no starvation occurs somewhere in the system, and helps validate the model.

ID	Property
MX.1	$\square(\text{take} \rightarrow (\neg \text{take} \mathcal{U} \text{release}))$
Liveness	$\square \diamond \text{take}$

## 7.5.2 VERIFICATION

Verification was performed by first validating the model using deadlock checking, and then model checking the properties as described above. As discussed in Section 5.3.3, vital for model checking is ensuring that no deadlocks occur, as these negatively affect the results. Furthermore, as all components are expected to be able to perform their function indefinitely, a deadlock would indicate an error in the specification. To this end, the components listed in Table 7.10 were individually checked for deadlocks.

However, this is not sufficient to validate the model. It is expected that all the

various subsystems remain live. The presence of a single, non-synchronizing component that is deadlock free implies the entire system is, even if some other part can no longer evolve to a different state. Therefore, both fairness constraints and specific safety properties are checked in addition. The fairness constraints ensure no component will starve another (which in the case study is unrealistic), the safety properties check that fair paths are reachable. The latter is achieved by specifying properties of the form  $\Box\text{-action}$  (e.g.,  $\Box\text{-take}$  for a mutex), and ensuring a counterexample can be found.

As COMPASS handles all the translation from SLIM to the backends, the validation and verification processes are automatic, requiring only the root component to be selected at start. The analysis was performed using the BDD engine of `NUXMV`, on a Core i7-930 @ 2.80 GHz system. For each individual component, both validation and verification were performed, and were all completed within 30 s of execution time. However, this was not possible for the complete model. Although the model was determined to be deadlock free, model checking timed out after running for 5 days.

Note that compared to the original BIP case study, the specification consists of fewer components. This is due to the I<sup>2</sup>C component implementation being simplified, and mutexes being absent for the `s128_1`, `s128_4` and `s128_5` components. However, the number of properties verified is higher, as those derived from the requirements are considered as well. The number of states differ significantly. However, depending on the component it can be more or less. This can be attributed to different interaction mechanics between SLIM and BIP, as well as the use of event data ports in the SLIM model for the I<sup>2</sup>Cbus.

## 7.6 DISCUSSION

The primary goal of the case study as described in this chapter was to provide a larger example for the COMPASS toolset with accompanying documentation. With that in mind, this chapter will hopefully provide enough of a base line to both be a platform for further development, as well as a guide for new users to get acquainted with the SLIM language. Additionally, it gives insight in the commonalities and differences between the BIP and SLIM languages.

Although most of the model could be translated from BIP into SLIM in a straightforward manner, some changes were made either because of differences in semantics or because a different expression of a concept in SLIM is more natural to the language (i.e., the use of event ports). In particular, but not solely, priority-based interactions were replaced by transition guards using additional state-based variables, and telecommand and I<sup>2</sup>C connections make use of event data ports instead of simple events. Finally, because the root component is a parameter of the verification process, components can easily be reused in different hierarchies.

The requirements provided in [102] were formalized as well. Though they only cover a subset of the functionality, their verification gives further assurance that

the model is correct. However, some inconsistencies had to be taken into account. The process of model verification is a straightforward task. Most of the properties from the case study could directly be translated, or be represented in a way that expresses their original intent. Translation and analysis of these properties is handled automatically. Properties specified on reused components are verified on all instances as well. However, the model has to be validated to ensure that it is specified correctly.

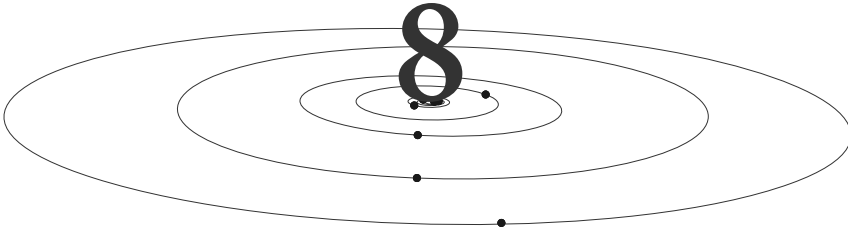
The SLIM model requires (manual) specification of fairness constraints. Since multiple components run in parallel, without such constraints verification takes into account the starvation of components, which is considered unrealistic. Therefore, for each such component, fairness constraints are added requiring them to show behavior infinitely often, excluding traces where starvation occurs. Addition of such fairness constraints require extra attention to validation of the model. Such properties exclude unwanted behavior, but may in addition also exclude behavior that *should* be considered, and that may even have invalidated properties were it so. To this end, a few properties are checked that deny desired behavior, such as to generate explicit counter-examples that prove such behavior exists.

Verifying the model proved to be a challenge which with the current state of the art (w.r.t. implementation) cannot fully be overcome. Although the individual subsystems could be analyzed, analysis of the complete CDMS system timed out, similar to what occurred in the BIP-based case study. Approaches based on compositional reasoning could provide a solution here, like the D-FINDER [17] deadlock analysis tool used for the BIP-based case study showed. However, COMPASS does not support compositional deadlock analysis. In addition, although it was considered to make use of compositional reasoning for verification, the CBA engine (cf. Section 5.3.4) used by COMPASS does not (yet) support fan-in connections, a concept used extensively throughout the case study. Therefore, although some prototype contracts were written, CBA was not performed. It makes, however, for interesting future work, as the verification performed shows that the composed model grows prohibitively large.

The BIP framework and COMPASS toolset provide two approaches to model specifications quite similar in nature. A good example is that the verification engine used by the BIP-based and SLIM-based case studies is the same, namely `nuXmv`. Though both platforms can be used for the verification of their respective models, the BIP framework shows a greater emphasis on code generation, whereas COMPASS has a stronger support for safety and reliability analysis. A variety of code generators exist for BIP, making it easy to deploy them on various hardware platforms.

COMPASS shows stronger integration for analysis. In particular, the SMV models provided in the original case study are generated by the BIP-toNuSMV tool [22], which does not support the complete BIP language — with restrictions similar to those of SLIM on interaction types — and requires manual insertion of fairness constraints and the properties to be verified (in SMV syntax). COMPASS provides a more integrated approach, as these steps are automated.

One aspect of the architecture pattern based design, that has not been considered, is the automatic derivation of properties. It is possible to derive such properties based on the actual implementation. The properties considered in this chapter were made by hand — though derived from the templates — to validate the result. Automated property generation may in particular be useful if the implementation is done by hand to ensure no errors are made. If this process itself is automated, the result can be ensured to be correct by construction. However, this requires a mapping from implementation to pattern, which is non-trivial. A possibility is to consider a graph-grammar based approach. This is left as future work.



## CONCLUSION

**N**OWADAYS the use of model-driven approaches is almost ubiquitous throughout the design of aerospace systems, as it can provide the necessary confidence in the correctness and reliability of systems. Formal methods provide an ideal platform to verify that the specification meets expectations, showing the system under development performs well not only under foreseen circumstances, but all those that can occur.

Such a platform is offered by COMPASS, which provides support for the analysis of the correctness, safety and performability of (space) systems. Central to these capabilities is the model that can be expressed using the SLIM language, which is covered in Chapter 2.

The *earlier* such methods can be applied, the more effective they are at detecting problems, and preventing them from affecting the processes later on. This starts at the requirements gathering phases. Chapter 3 considered the problem of the formal specification and verification of requirements. Various alternatives have been explored, most notably pattern-based and design attribute-based approaches. The latter has been implemented in the CSSP, which is one of the main outcomes of the CATSY project.

As projects are further developed, the design and specifications get further refined, resolving the functionality of systems by the definition of subsystems, with increasingly fine-grained detail. This holds for requirements as well as the (interfaces of the) systems themselves. Contract-based approaches naturally fit in with this pattern, offering both the possibility for tracking the refinement as the project develops, as well as compositional analysis methods, which, among other advantages, can avoid or mitigate the effects of the state space explosion problem. The refinement of requirements has been treated in Chapter 3, with the implementation of the contract-based analysis methods of COMPASS discussed in Chapter 5.

To tackle probabilistic analysis of stochastic-hybrid systems, a need specifically driven by the HASDEL project, the use of statistical model checking (SMC) has been investigated in Chapter 4. Although very promising, the use in the context of COMPASS does come with a number of caveats that need to be taken into account. In particular, certain choices have to be made with regard to underspecification, which is reflected by the SLIMSM tool developed as part of the process. Here, there is a clear engineering trade-off visible: On the one hand, SMC offers the ability to target a wide range of formalisms with a cost that does not depend on the state space size. On the other hand, numerical approaches can provide much greater precision and accuracy, and are better at detecting rare events. Neither approach replaces the other, but can be seen as complementary.

The COMPASS toolset wraps all these parts together, providing a platform where multiple formal techniques can be used together. Since its inception, it has been in continuous development throughout various projects driven both by the ESA and the EU. Its major strength is its ability to provide various (orthogonal) functions w.r.t. reliability and safety aspects of systems, as described in Chapter 5. It is this toolset that defines the basis on which the work in this thesis is founded.

Finally, the drive for applying COMPASS to practical problems led to the results of Chapters 6 and 7. In order to develop a demonstrator for the COMPASS toolset, both a translation from SLIM to Simulink, as well as a model of a (small) satellite were discussed in Chapter 6. Chapter 7 details a case study performed earlier in the context of the BIP framework, with the intent of providing a larger, accessible model for future use.

## 8.1 CHALLENGES AND FUTURE WORK

Various points of possible improvement can be found for future research ventures were presented throughout the previous chapters. They can be found at the end of each chapter, of which the most crucial ones are briefly restated here.

Though based on the existing AADL standard, the SLIM language has some features tailored specifically for the purpose of verifying system behavior, which cause it to not be fully compatible with AADL. Herein lies an opportunity for improvement, as full compatibility provides the option to share the language with other tooling, as well as making it easier to adopt. Alternatively, the COMPASS toolset itself could be altered to accept other modeling languages as well, provided they are backed by compatible formal semantics.

Requirements elicitation, specification and formalization are all fields of their own, and future work remains necessary in all fields. The design-attribute based approach of the CSSP has yet to be proven, and performing a full case study using it is most certainly desirable. In particular, it is expected that it does not provide a complete coverage of all relevant requirement types, and those gaps have to be found and closed.

The same holds for the COMPASS toolset itself: Further and more complete case studies help further evaluate it. Previous case studies have been performed, but the models they produced are not accessible. The case study presented in Chapter 7 provides a good start in this direction, but does not cover the full extent of what COMPASS has to offer. Expanding it with richer behavior (such as timed specifications), error models, contracts and CSSP property specifications would improve this. This may also help bring COMPASS to a higher technology readiness level, making it more likely to be adopted by the industry.

The use of formal approaches to tackle the problems described in this thesis allows for rigorous verification and validation. However, it leads to two problems, which have to be addressed to make it more usable in practice: Some of its functions require a detailed understanding of underlying concepts, and some functions may (unexpectedly) require a significant amount of resources, including time, to complete. As an example of the prior, correctness analysis relies on the fact the model is free from deadlocks, time divergence and specific types of Zeno behavior. Although tools exist to check for this, COMPASS still requires the user to be aware of this. Ideally, this is abstracted, such that an engineer does not have to be concerned with such problems. The high demand for resources is apparent, in particular with larger models, which may cause analysis to seemingly never end, or consume all available memory before terminating. Such issues may be addressed at the input level, perhaps by restricting the language in certain way, or by the development of new, and more efficient, techniques.



# A

## LOGICS

The following sections provide a formal definition of the logics that can be used for the property taxonomy. For each logic, the grammar is given in EBNF form. The symbols  $\varphi$  and  $\psi$  refer to subformulas, formed by their corresponding rules in the grammar. For each rule in the grammar, the semantic meaning is given in the table below. Many logics have various operators in common, albeit with differences in semantics, but identical intuition. An overview of these operators is given in the following table:

Operator	Meaning
$a$	Atomic proposition, which is generally a label of a state or transition. For example, $processor\_usage < 70$ .
$\wedge$	Boolean and
$\neg$	Boolean negation
$\mathcal{U}$	Until operator, indicating the formula on the left hand side must hold until the formula on the right hand side occurs. In particular, the formula does not hold if the formula never occurs.
$\square$	Always operator, indicating the formula must always hold.
$\diamond$	Eventually operator, indicating the formula must eventually hold (at least once).
$\circ$	Next operator, meaning the formula must hold in the next state.
$\mathcal{W}$	Weak until, similar to until, but does not require the formula on the right hand side to eventually occur.

In all cases, the grammar provides only a basic set of operators. Other common operators can be derived from these, for instance by the application of Boolean logic such as DeMorgan's Law. Below is a table with a set of such operators. Although not directly defined in each grammar, they can be composed of operators that do occur in the grammar.

For the Boolean operators, the usual equivalences can be used, e.g.

Boolean operator	Definition
$\varphi_1 \vee \varphi_2$	$\neg(\neg\varphi_1 \wedge \neg\varphi_2)$
$\varphi_1 \rightarrow \varphi_2$	$\neg\varphi_1 \vee \varphi_2$
$\varphi_1 \leftrightarrow \varphi_2$	$(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$

For the various path operators, the following equivalences can be used:

Path operator	Definition
$\diamond\varphi$	$\top \mathcal{U} \varphi$
$\square\varphi$	$\neg\diamond\neg\varphi$
$\varphi_1 \mathcal{W} \varphi_2$	$(\varphi_1 \mathcal{U} \varphi_2) \vee \square\varphi_1$

## A.1 LINEAR TEMPORAL LOGIC

Linear temporal logic (LTL) [116] is a temporal logic that can reason over paths in a qualitative setting. Linear temporal logic (LTL) supports a number of state and path operators according to the following grammar:

$$\varphi := a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \circ\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

Here  $a$  is an atomic proposition (AP), and  $\varphi$  an LTL formula.

The semantics of LTL are defined by the following satisfaction relations ( $\models$ ). If  $\sigma = A_0A_1A_2\dots$  is a path, then the following hold:

$$\begin{aligned} \sigma \models a & \quad \text{iff } a \in A_0 \\ \sigma \models \varphi_1 \wedge \varphi_2 & \quad \text{iff } \sigma \models \varphi_1 \wedge \sigma \models \varphi_2 \\ \sigma \models \neg\varphi & \quad \text{iff } \sigma \not\models \varphi \\ \sigma \models \circ\varphi & \quad \text{iff } \sigma[1\dots] = A_1A_2\dots \models \varphi \\ \sigma \models \varphi_1 \mathcal{U} \varphi_2 & \quad \text{iff } \exists j \in \mathbb{N}_0. \sigma[j\dots] \models \varphi_2 \wedge \forall i \in [0..j]. \sigma[i\dots] \models \varphi_1 \end{aligned}$$

## A.2 COMPUTATION TREE LOGIC

Computation tree logic (CTL)[49] is a logic similar to LTL. However, it reasons over trees of paths, i.e., it is a branching-time logic. Its grammar is similar to that of LTL, however, path operators are prefixed either with  $\forall$  (reasoning over all following paths) or  $\exists$  (reasoning over some following path). The following describes the computation tree logic (CTL) grammar. Here,  $\varphi$  are valid CTL formulas, and  $\psi$  unqualified path operators.

$$\begin{aligned} \varphi & := a \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid \exists\psi \mid \forall\psi \\ \psi & := \circ\varphi \mid \varphi_1 \mathcal{U} \varphi_2 \end{aligned}$$

The semantics of CTL are not based on paths, but rather trees. Whereas LTL consid-

ers a single (infinite) path, CTL considers all branches reachable from a given state. That is, given a state  $s$ , the operator  $\text{Paths}(s)$  gives all the (infinite) paths reachable from  $s$ . Based on this, a CTL formula is satisfied according to the following rules:

$$\begin{array}{ll}
s \models a & \text{iff } a \in L(s) \\
s \models \varphi_1 \wedge \varphi_2 & \text{iff } s \models \varphi_1 \wedge s \models \varphi_2 \\
s \models \neg\varphi & \text{iff } s \not\models \varphi \\
s \models \exists\varphi & \text{iff } \exists \pi \in \text{Paths}(s). \pi \models \varphi \\
s \models \forall\varphi & \text{iff } \forall \pi \in \text{Paths}(s). \pi \models \varphi \\
\pi \models \bigcirc\psi & \text{iff } \pi[1] \models \psi \\
\pi \models \psi_1 \mathcal{U} \psi_2 & \text{iff } \exists j \in \mathbb{N}_0. \pi[j] \models \psi_2 \wedge \forall i \in [0..j). \pi[i] \models \psi_1
\end{array}$$

### A.3 PROBABILISTIC CTL

Probabilistic computation tree logic (PCTL) [73] can be considered the probabilistic variant of CTL. Probabilistic computation tree logic (PCTL) replaces the path quantifiers of CTL with a probabilistic operator that compares the probability of the given subformula against some threshold. Its grammar is defined as:

$$\begin{array}{l}
\varphi := a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid P_{\bowtie p}(\psi), \text{ where } \bowtie \in \{<, \leq, >, \geq\} \text{ and } p \in [0, 1] \\
\psi := \varphi_1 \mathcal{U}^{\leq t} \varphi_2, \text{ with } t \in \mathbb{N}^+ \cup \{\infty\}.
\end{array}$$

Here,  $P_{\bowtie p}$  takes the probabilistic measure of the set of paths satisfying the given subformula, and compares it against  $p$  using the relation  $\bowtie$ . The path operator can be step-bounded, considering only paths with a length up to that bound. Note that the  $\bigcirc$  operator can be derived from  $\mathcal{U}$  with a step bound of one. The satisfaction rules for a given state  $s$  then become:

$$\begin{array}{ll}
s \models a & \text{iff } a \in L(s) \\
s \models \neg\varphi_1 & \text{iff } s \not\models \varphi_1 \\
s \models \varphi_1 \wedge \varphi_2 & \text{iff } s \models \varphi_1 \wedge s \models \varphi_2 \\
s \models P_{\bowtie p}(\psi) & \text{iff } \Pr[s \models \psi] \bowtie p \\
\pi \models \varphi_1 \mathcal{U}^t \varphi_2 & \text{iff } \exists j < u. \pi@j \models \varphi_2 \wedge \forall i \in [0..j). \pi@i \models \varphi_1
\end{array}$$

### A.4 CONTINUOUS STOCHASTIC LOGIC

Continuous stochastic logic (CSL) [12] is similar to PCTL, but is defined for continuous intervals rather than discrete ones. It is defined using the following grammar:

$$\begin{array}{l}
\varphi := a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid P_{\bowtie p}(\psi), \text{ where } \bowtie \in \{<, \leq, >, \geq\} \text{ and } p \in [0, 1] \\
\psi := \bigcirc^{\leq t} \varphi \mid \varphi_1 \mathcal{U}^{\leq t} \varphi_2, \text{ with } t \in \mathbb{R}^+ \cup \{\infty\}.
\end{array}$$

The semantics are as follows:

$s \models a$	iff $a \in L(s)$
$s \models \neg\varphi_1$	iff $s \not\models \varphi_1$
$s \models \varphi_1 \wedge \varphi_2$	iff $s \models \varphi_1 \wedge s \models \varphi_2$
$s \models P_{\bowtie p}(\varphi)$	iff $\Pr[s \models \varphi] \bowtie p$
$\pi \models \circ \psi$	iff $\pi[1] \models \psi$
$\pi \models \psi_1 \mathcal{U} \psi_2$	iff $\exists j \in \mathbb{N}_0. \pi[j] \models \psi_2 \wedge \forall i \in [0..j]. \pi[i] \models \psi_1$
$\pi \models \psi_1 \mathcal{U} [l, u] \psi_2$	iff $\exists j \in [l..u]. \pi @ j \models \psi_2 \wedge \forall i \in [0..j]. \pi @ i \models \psi_1$

## A.5 METRIC TEMPORAL LOGIC

Metric temporal logic (MTL) [94] is a logic that permits reasoning over timed systems. The path operator of metric temporal logic (MTL) is parameterized in the amount of time that is allowed to pass for all the paths that it considers. Its grammar is specified as:

$$\varphi := a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \varphi_1 \mathcal{U}^I \varphi_2, \text{ where } I = [l, u] \text{ with } l, u \in \mathbb{R} \wedge l < u$$

The semantics of MTL are defined by the following rules. Here,  $f : \mathbb{R}_+ \rightarrow 2^P$  is a function that maps a time point to the atomic propositions that hold true at that point in time, with  $f^t(s) = f(s + t)$ :

$$\begin{array}{ll} f \models a & \text{iff } f(s) = a \\ f \models \varphi_1 \wedge \varphi_2 & \text{iff } f \models \varphi_1 \wedge f \models \varphi_2 \\ f \models \neg\varphi & \text{iff } f \not\models \varphi \\ f \models \varphi_1 \mathcal{U}^I \varphi_2 & \text{iff } \exists j \in I. f^j \models \varphi_2 \wedge \forall k \in (0, j). f^k \models \varphi_1 \end{array}$$

## A.6 SLIM LOGICAL EXPRESSIONS

The expression syntax of SLIM follows that of most other languages, with infix binary operators to compare values, perform Boolean operations or arithmetic operations, and various unary operators with similar functions. The complete specification of these expressions can be found in [SLIMv3].

SLIM expressions can occur both in the model, as well as properties. In the model, these expressions refer to variables considering only a single (the current) configuration. These expressions occur as transition guards, in effects, mode invariants and data flows.

Within properties, the supported expression syntax is extended. Some special variables exist that properties may refer to, but are not explicitly defined in the model. These are **mode**, which refers to the active mode of the component the expressions is defined for, **error**, which refers to the current error state of a component if it has an associated error model, and **time**, which is the current amount of time that has passed since the initial state (for models that are timed).

Furthermore, operators are supported that can refer to past or future configurations, allowing them to be mapped to corresponding temporal operators in the aforementioned logics. These functions and their meaning are listed below.

The following table lists the built-in function operators that can be used in SLIM properties.

Operator	Function
<b>data</b> ( $d$ )	The data value of an event data port as it occurs. The value is undefined when the event is not occurring.
<b>last_data</b> ( $d$ )	The data value of an event data port the last time it was triggered. The value is undefined before the first event occurring.
<b>next</b> ( $x$ )	The value of variable $x$ at the next state (not to be confused with the LTL operator $\circ$ ).
<b>change</b> ( $x$ )	True when the value of $x$ changes going to the next state.
<b>rise</b> ( $b$ )	True when the Boolean expression $b$ becomes true in the next state.
<b>fall</b> ( $b$ )	True when the Boolean expression $b$ becomes false in the next state.
<b>time_until</b> ( $b$ )	The time until Boolean expression $b$ becomes true starting from the current state.
<b>time_since</b> ( $b$ )	The time since Boolean expression $b$ became true up to the current state.

**SLIM temporal expressions** Aside from the state-based operators mentioned above, SLIM supports a number of temporal operators as listed below.

Operator	Function	Formula
<b>always</b> $\varphi$	Whether $\varphi$ holds true in all following states.	$\Box\varphi$
<b>never</b> $\varphi$	Whether $\varphi$ never holds true in all following states.	$\Box\neg\varphi$
<b>historically</b> $\varphi$	Whether $\varphi$ held true in all past states.	
<b>in the future</b> $\varphi$	Whether $\varphi$ holds true in some following state.	$\Diamond\varphi$
<b>in the past</b> $\varphi$ ,		
<b>once</b> $\varphi$	Whether $\varphi$ held true in some past state.	
<b>X</b> $\varphi$ , <b>then</b> $\varphi$	Whether $\varphi$ holds true in the next state.	$\circ\varphi$
<b>Y</b> $\varphi$ ,		
<b>previously</b> $\varphi$	Whether $\varphi$ held true in the previous state.	
$\varphi_1$ <b>until</b> $\varphi_2$	True if eventually $\varphi_2$ holds true, and $\varphi_1$ holds true until then.	$\varphi_1 \mathcal{U} \varphi_2$

$\varphi_1$ <b>releases</b> $\varphi_2$	True if and only if $\varphi_2$ holds up to and including the point where $\varphi_1$ holds true. If $\varphi_1$ never occurs, $\varphi_2$ must hold true in all following states.	$\neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2)$
$\varphi_1$ <b>since</b> $\varphi_2$	True if and only if $\varphi_2$ held true since past state $t$ , after which $\varphi_1$ held true up to and including the current state.	
$\varphi_1$ <b>triggered</b> $\varphi_2$	True if and only if $\varphi_1$ held true since past state $t$ , at which point $\varphi_2$ held true up to and including the current state. Alternatively, if $\varphi_1$ held never true, $\varphi_2$ held true up to and including the current state.	
<b>ET</b> ( $\varphi$ )	The expected time until $\varphi$ becomes true.	
<b>LRA</b> ( $\varphi$ )	The long-run average fraction of time $\varphi$ is true.	

The last two properties are defined using probabilistic semantics, being either the *expected time (ET)* or *long-run average (LRA)*. These properties are based on the probabilistic semantics of SLIM as discussed in Chapter 2. Both metrics are explained in more detail in [70]. The ET metric is determined by the mean sojourn time of the states preceding those specified by the property, and considers paths of finite length. The LRA is determined by the fraction of time spent in the states specified by the property. For this, only paths of infinite length are considered (such an average does not exist for paths of finite length).

## A.7 OPERATOR PRECEDENCE

The precedence of the operators used is given by the table below, in order of descending precedence.

- |  |                      |
|--|----------------------|
| 1. Unary operators ( $\neg, \diamond, \circ$ etc.)                 | 4. $\wedge$          |
| 2. Binary relational operators<br>( $=, <, \geq$ etc.)             | 5. $\vee$            |
| 3. Binary temporal operators<br>( $\mathcal{U}, \mathcal{W}$ etc.) | 6. $\rightarrow$     |
|  | 7. $\leftrightarrow$ |

Binary relational operators are left-associative, the temporal operators are right associative.

# B

## PROPERTY PATTERNS

### B.1 GRAMMAR

COMPASS supports a textual representation for property patterns that can directly be embedded in SLIM specifications as AADL properties. Such patterns are parsed automatically when loading the specification and are then processed into formal properties.

$\langle pattern \rangle ::= \langle scope \rangle ' , ' \langle pattern \rangle (' . ')?$

$\langle scope \rangle ::= 'Globally' | 'Before' \langle prop \rangle | 'After' \langle prop \rangle$   
 $| 'Between' \langle prop \rangle 'and' \langle prop \rangle | 'After' \langle prop \rangle 'until' \langle prop \rangle$

$\langle pattern \rangle ::= \langle occurrence \rangle | \langle order \rangle$

$\langle occurrence \rangle ::= \langle universality \rangle | \langle absense \rangle | \langle existence \rangle | \langle recurrence \rangle$

$\langle order \rangle ::= \langle precedence \rangle | \langle until \rangle | \langle response \rangle$

$\langle universality \rangle ::= 'it is always the case that' \langle prop \rangle ('holds')?$   
 $((time))? ((probability))?$

$\langle absense \rangle ::= 'it is never the case that' \langle prop \rangle ('holds')?$   
 $((time))? ((probability))?$

$\langle existence \rangle ::= \langle prop \rangle ('holds')? 'eventually'$   
 $((time))? ((probability))?$

$\langle recurrence \rangle ::= \langle prop \rangle ('holds')? 'repeatedly'$   
 $('every' (timeUnit))? ((probability))?$

$\langle precedence \rangle ::= \text{if } \langle prop \rangle \text{ ('holds')}? \text{ 'then it must be the case that' } \langle prop \rangle \text{ ('has occurred')}? \text{ 'before' } (\langle time \rangle)? (\langle probability \rangle)?$

$\langle until \rangle ::= \langle prop \rangle \text{ ('holds')}? \text{ ('without interruption')}? \text{ 'until' } \langle prop \rangle \text{ 'holds' } (\langle time \rangle)? (\langle probability \rangle)?$

$\langle response \rangle ::= \text{'if' } \langle prop \rangle \text{ ('has occurred')}? \text{ 'then in response' } \langle prop \rangle \text{ ('eventually holds')}? (\langle time \rangle)? (\langle probability \rangle)?$

$\langle time \rangle ::= \text{'within' } \langle timeUnit \rangle \text{ | 'after' } \langle timeUnit \rangle \text{ | 'between' } \langle timeUnit \rangle \text{ 'and' } \langle timeUnit \rangle$

$\langle timeUnit \rangle ::= \langle number \rangle \text{ ('msec' | 'sec' | 'min' | 'hour' | 'day')}?$

$\langle probability \rangle ::= \text{'with probability' } (<' | '<=' | '>' | '>=') \langle number \rangle$

$\langle number \rangle ::= \text{Numeric SLIM expression}$

$\langle prop \rangle ::= \text{'{' SLIM expression '}'}$

## B.2 FORMULAS

For the translation of pattern specifications to formal properties, the following mappings are used. Here,  $\varphi$  and  $\psi$  refer to the propositions specified for the various pattern classes. The propositions  $\rho$  and  $\omega$  are those specified for the scopes. Time bounds are expressed in terms of a lower bound  $l$  and an upper bound  $u$ . The probabilistic patterns use  $p$  to indicate the queried probability, and  $\bowtie$  the relation to compare against —  $\bar{\bowtie}$  being the inverse relation where applicable.

Not all combinations of scopes, classes and property types are supported. Where this is not the case is indicated. In some cases this is simply a limitation of COMPASS and may be implemented in the future.

### B.2.1 EXISTENCE

Eventually  $\varphi$

*Qualitative*

Globally	$\diamond \varphi$
Before $\omega$	$\neg \omega \mathcal{W} (\varphi \wedge \neg \omega)$
After $\rho$	$\square \neg \rho \vee (\diamond \rho \wedge \diamond \varphi)$
Between $\rho$ and $\omega$	$\square (\rho \wedge \neg \omega) \rightarrow (\neg \omega \mathcal{W} (\varphi \wedge \neg \omega))$
After $\rho$ until $\omega$	$\square (\rho \wedge \neg \omega) \rightarrow (\neg \omega \mathcal{U} (\varphi \wedge \neg \omega))$

*Timed*

Globally	$\diamond^{[l,u]} \varphi$
Before $\omega$	$\neg \omega \mathcal{W}^{[l,\infty)} (\varphi \wedge \neg \omega)$
After $\rho$	$\square \neg \rho \vee (\diamond \rho \wedge (\diamond^{[l,u]} \varphi))$
Between $\rho$ and $\omega$	$\square((\rho \wedge (\square^{[0,l]} \neg \omega)) \wedge (\diamond^{[l,\infty)} \omega)) \rightarrow (\neg \omega \mathcal{W}^{[l,u]} (\varphi \wedge \neg \omega))$
After $\rho$ until $\omega$	$\square(\rho \wedge (\square^{[0,l]} \neg \omega)) \rightarrow (\neg \omega \mathcal{U}^{[l,u]} (\varphi \wedge \neg \omega))$

*Probabilistic*

Globally	$\Pr_{\times p}[\diamond^{[l,u]} \varphi]$
Before $\omega$	$\Pr_{\times p}[\neg \omega \mathcal{W}^{[l,\infty)} (\varphi \wedge \neg \omega)]$
After $\rho$	$\Pr_{\geq 1.0}[\neg \rho \mathcal{W} (\rho \wedge (\Pr_{\times p}[\diamond^{[l,u]} \varphi]))]$
Between $\rho$ and $\omega$	(Not supported)
After $\rho$ until $\omega$	(Not supported)

## B.2.2 UNIVERSALITY

Always  $\varphi$ *Qualitative*

Globally	$\square \varphi$
Before $\omega$	$\diamond \omega \rightarrow (\varphi \mathcal{U} \omega)$
After $\rho$	$\square \rho \rightarrow \square \varphi$
Between $\rho$ and $\omega$	$\square((\rho \wedge \neg \omega) \wedge \diamond \omega) \rightarrow (\varphi \mathcal{U} \omega)$
After $\rho$ until $\omega$	$\square(\rho \wedge \neg \omega) \rightarrow (\varphi \mathcal{W} \omega)$

*Timed*

Globally	$\square^{[l,u]} \varphi$
Before $\omega$	$(\diamond^{[l,\infty)} \omega) \rightarrow ((\varphi \mathcal{U}^{[l,u]} \omega) \vee (\square^{[l,u]} \varphi))$
After $\rho$	$\square \rho \rightarrow (\square^{[l,u]} \varphi)$
Between $\rho$ and $\omega$	$\square((\rho \wedge (\square^{[0,l]} \neg \omega)) \wedge (\diamond^{[l,\infty)} \omega)) \rightarrow ((\varphi \mathcal{U}^{[l,u]} \omega) \vee (\square^{[l,u]} \varphi))$
After $\rho$ until $\omega$	$\square(\rho \wedge (\square^{[0,l]} \neg \omega)) \rightarrow (\varphi \mathcal{W}^{[l,u]} \omega)$

*Probabilistic*

Globally	$\Pr_{\times p}[\square^{[l,u]} \varphi]$
Before $\omega$	$\Pr_{\times p}[(\diamond^{[l,\infty)} \omega) \rightarrow ((\varphi \mathcal{U}^{[l,u]} \omega) \vee (\Pr_{\geq 1.0}[\square^{[l,u]} \varphi]))]$
After $\rho$	$\Pr_{\geq 1.0}[\square \rho \rightarrow (\Pr_{\times p}[\square^{[l,u]} \varphi])]$
Between $\rho$ and $\omega$	$\Pr_{\geq 1.0}[\square(\rho \wedge (\Pr_{\geq 1.0}[\square^{[0,l]} \neg \omega])) \rightarrow (\Pr_{\times p}[(\diamond^{[l,\infty)} \omega) \rightarrow ((\varphi \mathcal{U}^{[l,u]} \omega) \vee (\Pr_{\geq 1.0}[\square^{[l,u]} \varphi])))]]$

After  $\rho$  until  $\omega$   $\Pr_{\geq 1.0}[\Box(\rho \wedge (\Pr_{\geq 1.0}[\Box^{[0,l]} \neg \omega])) \rightarrow (\Pr_{\neq p}[\varphi \mathcal{W}^{[l,u]} \omega])]$

### B.2.3 ABSENSE

Never  $\varphi$

#### Qualitative

Globally  $\Box \neg \varphi$   
 Before  $\omega$   $\Diamond \omega \rightarrow (\neg \varphi \mathcal{U} \omega)$   
 After  $\rho$   $\Box \rho \rightarrow \Box \neg \varphi$   
 Between  $\rho$  and  $\omega$   $\Box((\rho \wedge \neg \omega) \wedge \Diamond \omega) \rightarrow (\neg \varphi \mathcal{U} \omega)$   
 After  $\rho$  until  $\omega$   $\Box(\rho \wedge \neg \omega) \rightarrow (\neg \varphi \mathcal{W} \omega)$

#### Timed

Globally  $\Box^{[l,u]} \neg \varphi$   
 Before  $\omega$   $(\Diamond^{[l,\infty)} \omega) \rightarrow ((\neg \varphi \mathcal{U}^{[l,u]} \omega) \vee (\Box^{[l,u]} \neg \varphi))$   
 After  $\rho$   $\Box \rho \rightarrow (\Box^{[l,u]} \neg \varphi)$   
 Between  $\rho$  and  $\omega$   $\Box((\rho \wedge (\Box^{[0,l]} \neg \omega)) \wedge (\Diamond^{[l,\infty)} \omega)) \rightarrow ((\neg \varphi \mathcal{U}^{[l,u]} \omega) \vee (\Box^{[l,u]} \neg \varphi))$   
 After  $\rho$  until  $\omega$   $\Box(\rho \wedge (\Box^{[0,l]} \neg \omega)) \rightarrow (\neg \varphi \mathcal{W}^{[l,u]} \omega)$

#### Probabilistic

Globally  $\Pr_{\neq p}[\Box^{[l,u]} \neg \varphi]$   
 Before  $\omega$   $\Pr_{\neq p}[(\Diamond^{[l,\infty)} \omega) \rightarrow ((\neg \varphi \mathcal{U}^{[l,u]} \omega) \vee (\Pr_{\geq 1.0}[\Box^{[l,u]} \neg \varphi]))]$   
 After  $\rho$   $\Pr_{\geq 1.0}[\Box \rho \rightarrow (\Pr_{\neq p}[\Box^{[l,u]} \neg \varphi])]$   
 Between  $\rho$  and  $\omega$   $\Pr_{\geq 1.0}[\Box(\rho \wedge (\Pr_{\geq 1.0}[\Box^{[0,l]} \neg \omega])) \rightarrow (\Pr_{\neq p}[(\Diamond^{[l,\infty)} \omega) \rightarrow ((\neg \varphi \mathcal{U}^{[l,u]} \omega) \vee (\Pr_{\geq 1.0}[\Box^{[l,u]} \neg \varphi]))])]$   
 After  $\rho$  until  $\omega$   $\Pr_{\geq 1.0}[\Box(\rho \wedge (\Pr_{\geq 1.0}[\Box^{[0,l]} \neg \omega])) \rightarrow (\Pr_{\neq p}[\neg \varphi \mathcal{W}^{[l,u]} \omega])]$

### B.2.4 RECURRENCE

Repeatedly  $\varphi$

#### Qualitative

Globally  $\Box \Diamond \varphi$   
 Before  $\omega$   $\Diamond \omega \rightarrow ((\Diamond \varphi \vee \omega) \mathcal{U} \omega)$   
 After  $\rho$   $\Box \rho \rightarrow (\Box \Diamond \varphi)$

Between $\rho$ and $\omega$	$\Box((\rho \wedge \neg\omega) \wedge \Diamond\omega) \rightarrow ((\Diamond\varphi \vee \omega) \mathcal{U} \omega)$
After $\rho$ until $\omega$	$\Box(\rho \wedge \neg\omega) \rightarrow ((\Diamond\varphi \vee \omega) \mathcal{W} \omega)$

*Timed*

Globally	$\Box\Diamond^{[0,u]}\varphi$
Before $\omega$	$\Diamond\omega \rightarrow ((\Diamond^{[0,u]}\varphi \vee \omega) \mathcal{U} \omega)$
After $\rho$	$\Box\rho \rightarrow (\Box\Diamond^{[0,u]}\varphi)$
Between $\rho$ and $\omega$	$\Box((\rho \wedge \neg\omega) \wedge \Diamond\omega) \rightarrow ((\Diamond^{[0,u]}\varphi \vee \omega) \mathcal{U} \omega)$
After $\rho$ until $\omega$	$\Box(\rho \wedge \neg\omega) \rightarrow ((\Diamond^{[0,u]}\varphi \vee \omega) \mathcal{W} \omega)$

*Probabilistic*

Globally	$\Pr_{\geq p}[\Box\Diamond^{[0,u]}\varphi]$
Before $\omega$	$\Pr_{\geq 1.0}[\Diamond\omega \rightarrow (\Pr_{\geq p}[(\Diamond^{[0,u]}\varphi \vee \omega) \mathcal{U} \omega])]$
After $\rho$	$\Pr_{\geq 1.0}[\Box\rho \rightarrow (\Pr_{\geq p}[\Box\Diamond^{[0,u]}\varphi])]$
Between $\rho$ and $\omega$	$\Pr_{\geq 1.0}[\Box((\rho \wedge \neg\omega) \wedge \Diamond\omega) \rightarrow (\Pr_{\geq p}[(\Diamond^{[0,u]}\varphi \vee \omega) \mathcal{U} \omega])]$
After $\rho$ until $\omega$	$\Pr_{\geq 1.0}[\Box(\rho \wedge \neg\omega) \rightarrow (\Pr_{\geq p}[(\Diamond^{[0,u]}\varphi \vee \omega) \mathcal{W} \omega])]$

## B.2.5 PRECEDENCE

 $\psi$  precedes  $\varphi$ *Qualitative*

Globally	$\neg\varphi \mathcal{W} \psi$
Before $\omega$	$\Diamond\omega \rightarrow (\neg\varphi \mathcal{U} (\psi \vee \omega))$
After $\rho$	$\Box\neg\rho \vee (\Diamond\rho \wedge (\neg\varphi \mathcal{W} \omega))$
Between $\rho$ and $\omega$	$\Box((\rho \wedge \neg\omega) \wedge \Diamond\omega) \rightarrow \neg\varphi \mathcal{U} (\psi \vee \omega)$
After $\rho$ until $\omega$	$\Box(\rho \wedge \neg\omega) \rightarrow (\neg\varphi \mathcal{W} (\psi \vee \omega))$

*Timed*

Globally	$\Box(\Diamond^{[u,u]}\varphi) \rightarrow (\Diamond^{[0,u-l]}\psi)$
Before $\omega$	$\Diamond\omega \rightarrow (((\Diamond^{[u,u]}\varphi) \rightarrow ((\Diamond^{[0,u-l]}\psi) \vee (\Diamond^{[0,u]}\omega))) \mathcal{U} \omega)$
After $\rho$	$\Box\rho \rightarrow (\Box(\Diamond^{[u,u]}\varphi) \rightarrow (\Diamond^{[0,u-l]}\psi))$
Between $\rho$ and $\omega$	$\Box((\rho \wedge \neg\omega) \wedge \Diamond\omega) \rightarrow (((\Diamond^{[u,u]}\varphi) \rightarrow ((\Diamond^{[0,u-l]}\psi) \vee (\Diamond^{[0,u]}\omega))) \mathcal{U} \omega)$
After $\rho$ until $\omega$	$\Box(\rho \wedge \neg\omega) \rightarrow (((\Diamond^{[u,u]}\varphi) \rightarrow (\Diamond^{[0,u-l]}\psi \vee \Diamond^{[0,u]}\omega)) \mathcal{W} \omega)$

*Probabilistic*

Globally	$\Pr_{\approx 1-p}[-\psi \mathcal{U}^{[l,u]}(\neg\psi \vee \varphi)]$
Before $\omega$	(Not supported)
After $\rho$	(Not supported)
Between $\rho$ and $\omega$	(Not supported)
After $\rho$ until $\omega$	(Not supported)

### B.2.6 RESPONSE

$\varphi$  is followed by  $\psi$

#### Qualitative

Globally	$\Box\varphi \rightarrow \Diamond\psi$
Before $\omega$	$\Diamond\omega \rightarrow ((\varphi \rightarrow (\neg\omega \mathcal{U}(\psi \vee \neg\omega))) \mathcal{U} \omega)$
After $\rho$	$\Box\rho \rightarrow (\Box\varphi \rightarrow \Diamond\psi)$
Between $\rho$ and $\omega$	$\Box(((\rho \wedge \neg\omega) \wedge \Diamond\omega) \rightarrow (\varphi \rightarrow (\neg\omega \mathcal{U}(\psi \wedge \neg\omega)))) \mathcal{U} \omega$
After $\rho$ until $\omega$	$\Box((\rho \wedge \neg\omega) \rightarrow (\varphi \rightarrow (\neg\omega \mathcal{U}(\psi \wedge \neg\omega)))) \mathcal{W} \omega$

#### Timed

Globally	$\Box\varphi \rightarrow (\Diamond^{[l,u]}\psi)$
Before $\omega$	$(\Diamond^{[l,\infty)}\omega) \rightarrow ((\varphi \rightarrow (\neg\omega \mathcal{U}^{[l,u]}(\psi \vee \neg\omega))) \mathcal{U} \omega)$
After $\rho$	$\Box\rho \rightarrow (\Box\varphi \rightarrow (\Diamond^{[l,u]}\psi))$
Between $\rho$ and $\omega$	$\Box(((\rho \wedge (\Box^{[0,l]}\neg\omega)) \wedge (\Diamond^{[l,\infty)}\omega)) \rightarrow (\varphi \rightarrow (\neg\omega \mathcal{U}^{[l,u]}(\psi \wedge \neg\omega)))) \mathcal{U} \omega$
After $\rho$ until $\omega$	$\Box((\rho \wedge (\Box^{[0,l]}\neg\omega)) \rightarrow (\varphi \rightarrow (\neg\omega \mathcal{U}^{[l,u]}(\psi \wedge \neg\omega)))) \mathcal{W} \omega$

#### Probabilistic

Globally	$\Pr_{\geq 1.0}[\Box\varphi \rightarrow (\Pr_{\approx p}[\Diamond^{[l,u]}\psi])]$
Before $\omega$	(Not supported)
After $\rho$	$\Pr_{\geq 1.0}[\neg\rho \mathcal{W}(\rho \wedge (\Pr_{\geq 1.0}[\Box\varphi \rightarrow (\Pr_{\approx p}[\Diamond^{[l,u]}\psi])])]$
Between $\rho$ and $\omega$	(Not supported)
After $\rho$ until $\omega$	(Not supported)

### B.2.7 RESPONSE-INVARIANCE

$\varphi$  is followed by  $\psi$  continuously

#### Qualitative

Globally	$\Box(\varphi \rightarrow \Box\psi)$
Before $\omega$	$\Diamond\omega \rightarrow ((\varphi \rightarrow (\Box\psi \wedge \neg\omega)) \mathcal{U} \omega)$

After $\rho$	$\Box(\rho \rightarrow (\Box\varphi \rightarrow \Box\psi))$
Between $\rho$ and $\omega$	$\Box(((\rho \wedge \neg\omega) \wedge \Diamond\omega) \rightarrow (\varphi \rightarrow (\Box\psi \wedge \neg\omega))) \mathcal{U} \omega$
After $\rho$ until $\omega$	$\Box((\rho \wedge \neg\omega) \rightarrow (\varphi \rightarrow (\Box\psi \wedge \neg\omega))) \mathcal{W} \omega$

*Timed*

Globally	$\Box(\varphi \rightarrow (\Box^{[l,u]}\psi))$
Before $\omega$	$(\Diamond^{[l,\infty)}\omega) \rightarrow ((\varphi \rightarrow (\Box^{[l,u]}\psi \wedge \neg\omega))) \mathcal{U} \omega$
After $\rho$	$\Box(\rho \rightarrow (\Box\varphi \rightarrow (\Box^{[l,u]}\psi)))$
Between $\rho$ and $\omega$	$\Box(((\rho \wedge (\Box^{[0,l]}\neg\omega)) \wedge (\Diamond^{[l,\infty)}\omega)) \rightarrow (\varphi \rightarrow (\Box^{[l,u]}\psi \wedge \neg\omega))) \mathcal{U} \omega$
After $\rho$ until $\omega$	$\Box((\rho \wedge (\Box^{[0,l]}\neg\omega)) \rightarrow (\varphi \rightarrow (\Box^{[l,u]}\psi \wedge \neg\omega))) \mathcal{W} \omega$

*Probabilistic*

Globally	$\Pr_{\geq 1.0}[\Box(\varphi \rightarrow (\Pr_{\bowtie\rho}[\Box^{[l,u]}\psi]))]$
Before $\omega$	$\Pr_{\geq 1.0}[(\varphi \rightarrow ((\Pr_{\bowtie\rho}[\Box^{[l,u]}\psi \wedge \neg\omega]) \vee (\Pr_{\geq 1.0}[\Box\neg\omega]))) \mathcal{W} \omega]$
After $\rho$	$\Pr_{\geq 1.0}[\neg\rho \mathcal{W} (\rho \wedge (\Pr_{\geq 1.0}[\Box\varphi \rightarrow (\Pr_{\bowtie\rho}[\Box^{[l,u]}\psi)]))]$
Between $\rho$ and $\omega$	$\Pr_{\geq 1.0}[\Box(\rho \wedge (\Pr_{\geq 1.0}[\Box^{[0,l]}\neg\omega])) \rightarrow (\Pr_{\geq 1.0}[(\varphi \rightarrow (\Pr_{\bowtie\rho}[\Box^{[l,u]}\psi \wedge \neg\omega]) \vee (\Pr_{\geq 1.0}[\Box\neg\omega])) \mathcal{W} \omega])]$
After $\rho$ until $\omega$	$\Pr_{\geq 1.0}[\Box(\rho \wedge (\Pr_{\geq 1.0}[\Box^{[0,l]}\neg\omega])) \rightarrow (\Pr_{\geq 1.0}[(\varphi \rightarrow (\Pr_{\bowtie\rho}[\Box^{[l,u]}\psi \wedge \neg\omega])) \mathcal{W} \omega])]$

## B.2.8 UNTIL

Until  $\psi$  occurs,  $\varphi$  holds

*Qualitative*

Globally	$\varphi \mathcal{U} \psi$
Before $\omega$	$\Diamond\omega \rightarrow ((\varphi \wedge \neg\omega) \mathcal{U} (\psi \vee \omega))$
After $\rho$	$\Box\rho \rightarrow (\varphi \mathcal{U} \psi)$
Between $\rho$ and $\omega$	$\Box(((\rho \wedge \neg\omega) \wedge \Diamond\omega) \rightarrow (\varphi \mathcal{U} \psi)) \mathcal{U} (\psi \wedge \neg\omega)$
After $\rho$ until $\omega$	$\Box(\rho \wedge \neg\omega) \rightarrow ((\varphi \mathcal{U} \psi) \mathcal{W} \omega)$

*Timed*

Globally	$\varphi \mathcal{U}^{[l,u]} \psi$
Before $\omega$	$(\Diamond^{[l,\infty)}\omega) \rightarrow ((\varphi \wedge \neg\omega) \mathcal{U}^{[l,u]} (\psi \vee \omega))$
After $\rho$	$\Box\rho \rightarrow (\varphi \mathcal{U}^{[l,u]} \psi)$
Between $\rho$ and $\omega$	$\Box((\rho \wedge \neg\omega) \wedge (\Diamond^{[l,\infty)}\omega)) \rightarrow ((\varphi \wedge \neg\omega) \mathcal{U}^{[l,u]} (\psi \vee \omega))$

After  $\rho$  until  $\omega$        $\Box(\rho \wedge (\Box^{[0,l]}-\omega)) \rightarrow ((\varphi \mathcal{U}^{[l,u]} \psi) \mathcal{W} \omega)$

202

*Probabilistic*

Globally	$\Pr_{\times\rho}[\varphi \mathcal{U}^{[l,u]} \psi]$
Before $\omega$	(Not supported)
After $\rho$	(Not supported)
Between $\rho$ and $\omega$	(Not supported)
After $\rho$ until $\omega$	(Not supported)

# C

## CSSP

This appendix provides the complete specification of the CSSP, including the properties defined therein and the formal specifications that can be derived from them. In addition, it lists the design attributes that were considered when designing the CSSP, and how they can be mapped to either CSSP properties or model elements.

### C.1 CSSP PROPERTY SET

The following table lists the properties defined in the CSSP property set. For each property, it is indicated to which model elements they may apply (in terms of SLIM), and the type of the property (i.e., which values they can take). For the property types, **aadlstring** is used to allow the use of SLIM expressions, which otherwise cannot be used as property values. The **Time** property type is used to specify time delays. If the SLIM model makes use of time units, such values are required to have a unit specified as well, and likewise for unit-less models it can be left out.

(List starts on the next page)

Property Name	Property Type : Applies To	Description
Change	data, data port : event (data) port	Port to monitor for change events
ModeInhibited	list of event (data) port : mode	Specify which events are inhibited in the mode this property is applied to.
ModeInvariant	aadlstring : mode	Invariant of the mode this property is applied to.
MonitorRange	range of aadlinteger : event data port	Acceptable range of values of the given (integer type) port.
MonitorResponse	event (data) port : event data port	Port that should be triggered in response to a change in the monitored port outside the acceptable interval.
MonitorDelay	Time : event data port	Permissible delay in the response.
MonitorEnabled	list of mode : event data port	Modes in which monitoring is enabled.
AlarmDelay	Time : event (data) port	Permissible delay between occurrence of FailureCondition and the triggering of the alarm port.
RecoveryDelay	Time : event (data) port	Permissible delay between triggering of the alarm port and recovery from FailureCondition.
Timeout	Time : event (data) port	Maximum delay before the timeout event port is to be triggered.
TimeoutReset	event (data) port : event (data) port	Event which will reset the timeout counter.
TimeoutCondition	list of mode : event (data) port	Modes in which the timeout property is monitored.
Function	aadlstring : (event) data port	Function describing permissible values of the port this property applies to.
InvariantRange	range of aadlinteger : (event) data port	Range describing permissible values of the port this property applies to.

Reaction	event (data) port : event (data) port	Port expected to trigger in reaction to an event on the port this property applies to.
ReactionCondition	list of mode : event (data) port	Modes in which the reaction property is monitored.
ReactionMinDelay	Time : event (data) port	Minimum reaction delay.
ReactionMaxDelay	Time : event (data) port	Maximum reaction delay.
PrecededBy	event (data) port : event (data) port	Port that must have triggered before an event on the port this property applies to occurs.
PrecededCondition	list of mode : event (data) port	Modes in which the precedence property is monitored.
PrecededMinDelay	Time : event (data) port	Minimum delay between events.
PrecededMaxDelay	Time : event (data) port	Maximum delay between events.
PeriodInterval	Time : event (data) port	Period between events on the port this property applies to.
PeriodOffset	Time : event (data) port	Time offset of the period from system start.
PeriodJitter	Time : event (data) port	Allowed deviation from the period interval.
PeriodEnabled	list of mode : event (data) port	Modes in which the period property is monitored.
ThroughputInput	event (data) port : event (data) port	Port, with periodic events, to compare event rate with.
ThroughputRatio	aadlinteger : event (data) port	Multiplier of expected event rate compared to other port.
Tolerance	aadlinteger : port	Total amount of events that are allowed to occur on the port this property applies to.
FailureCondition	list of mode : system, device, process, thread, thread group, bus, processor, memory, port	Modes in which the component the property applies to is considered failed.

## C.2 CSSP FORMAL PROPERTIES

206

The following table lists all the formal properties that can be derived from the CSSP properties specified in the previous section, along with the CSSP properties that need to be defined in order to make it realizable. Each formal property is parameterized with some element of the model to which this formal property applies, the possible type of which is listed in the second column.

Property Name Applies To	Description
PersistentProperty( $p$ ) $p \in$ data component, data port	Specifies that the value of $p$ changes only on the <b>Change</b> ( $p$ ) event.
ModeInhibitedProperty( $m$ ) $m \in$ mode	Specifies that the events in <b>ModeInhibited</b> ( $m$ ) cannot occur in mode $m$ .
ModeInvariantProperty( $m$ ) $m \in$ mode	Specifies a generic invariant for mode $m$ based on <b>ModeInvariant</b> ( $m$ ).
MonitorProperty( $p$ ) $p \in$ event data port	where $u =$ <b>MonitorDelay</b> ( $p$ ). Specifies that the event <b>MonitorResponse</b> ( $p$ ) is fired if the value of $p$ falls outside the specified <b>MonitorRange</b> ( $p$ ).
CompleteAlarmProperty( $p$ ) $p \in$ event (data) port	where $u =$ <b>AlarmDelay</b> ( $p$ ). Specifies that if failure configuration <b>FailureCondition</b> ( $p$ ) is entered, the alarm event $p$ follows.
CorrectAlarmProperty( $p$ ) $p \in$ event (data) port	where $u =$ <b>AlarmDelay</b> ( $p$ ). Specifies that if the alarm event $p$ occurs, it was preceded by entering the failure configuration <b>FailureCondition</b> ( $p$ ).
RecoveryProperty( $p$ ) $p \in$ event (data) port	where $u =$ <b>RecoveryDelay</b> ( $p$ ). Specifies that upon event $p$ , eventually the failure configuration <b>FailureCondition</b> ( $p$ ) is recovered.
CompleteTimeoutProperty( $p$ ) $p \in$ event (data) port	where $u =$ <b>Timeout</b> ( $p$ ). Specifies that if $p$ does not occur within <b>Timeout</b> ( $p$ ), the alarm <b>TimeoutReset</b> ( $p$ ) must occur.
CorrectTimeoutProperty( $p$ ) $p \in$ event (data) port	where $u =$ <b>Timeout</b> ( $p$ ). Specifies that if the alarm <b>TimeoutReset</b> ( $p$ ) occurs, the event $p$ did not occur.
FunctionProperty( $p$ ) $p \in$ data	Specifies the value of $p$ remains within the associated function <b>Function</b> ( $p$ ) (an expression).
FunctionProperty( $p$ ) $p \in$ event data port	Specifies the value of $p$ remains within the associated function <b>Function</b> ( $p$ ) (an expression).

InvariantProperty( $p$ ) $p \in$ (event) data port	Specifies the value of $p$ remains within the range of values specified by <b>InvariantRange</b> ( $p$ ).
ReactionProperty( $p$ ) $p \in$ event (data) port	Specifies the event $p$ is followed by <b>Reaction</b> ( $p$ ) provided the mode is in <b>ReactionCondition</b> ( $p$ ). The permissible delay can optionally be specified using <b>ReactionMinDelay</b> ( $p$ ) and <b>ReactionMaxDelay</b> ( $p$ ).
PrecededByProperty( $p$ ) $p \in$ event (data) port	Specifies the event $p$ is preceded by <b>PrecededBy</b> ( $p$ ) provided the mode is in <b>PrecededCondition</b> ( $p$ ). The permissible delay can optionally be specified using <b>PrecededMinDelay</b> ( $p$ ) and <b>PrecededMaxDelay</b> ( $p$ ).
PeriodProperty( $p$ ) $p \in$ event (data) port	where $u =$ <b>PeriodInterval</b> ( $p$ ), $v =$ <b>PeriodOffset</b> , $j =$ <b>PeriodJitter</b> , $enabled = mode \in$ <b>PeriodEnabled</b> . Specifies the event $p$ occurs within the specified period and optional offset.
ThroughputRatio( $p$ ) $p \in$ event (data) port	Specifies the throughput of event $p$ as a ratio of the throughput of <b>ThroughputInput</b> ( $p$ ).
ToleranceProperty( $p$ ) $p \in$ port	Specifies the tolerated number of failure events <b>Tolerance</b> ( $p$ ). This is generally an assumption for other properties.
MTTF( $x$ ) $x \in$ event port, component	The expected time (mean time) until <b>FailureCondition</b> ( $x$ ) holds.
MTTR( $x$ ) $x \in$ event port, component	The expected time (mean time) until <b>FailureCondition</b> ( $x$ ) no longer holds.
Availability( $s$ ) $s \in$ system	The availability specified as the <i>long-run average</i> of $s$ being in a nominal mode.

---

### C.3 CSSP FORMAL DEFINITIONS

208

Here the definitions of the formal properties — specified in MTL (cf. section A.5) — are listed that can be derived from the CSSP properties.

$$\text{PersistentProperty}(p) = \square(\text{change}(p) \rightarrow \mathbf{Change}(p))$$

$$\text{ModeInhibitedProperty}(m) = \square(\text{mode} = m \rightarrow \bigwedge_{e \in \mathbf{ModeInhibited}(m)} \neg e)$$

$$\text{ModeInvariantProperty}(m) = \square(\text{mode} = m \rightarrow \mathbf{ModeInvariant}(m))$$

$$\begin{aligned} \text{MonitorProperty}(p) = \square((p \wedge \text{mode} \in \mathbf{MonitorEnabled}(p) \wedge \\ (\text{data}(p) \notin \mathbf{MonitorRange}(p))) \rightarrow \\ \diamond_{\leq u} \mathbf{MonitorResponse}(p)) \end{aligned}$$

$$\begin{aligned} \text{CompleteAlarmProperty}(p) = \square(\text{rise}(\text{mode} \in \mathbf{FailureCondition}(p)) \rightarrow \\ \diamond_{\leq u} p) \end{aligned}$$

$$\begin{aligned} \text{CorrectAlarmProperty}(p) = \square(p \rightarrow \\ \text{O}_{\leq u} \text{rise}(\text{mode} \in \mathbf{FailureCondition}(p))) \end{aligned}$$

$$\text{RecoveryProperty}(p) = \square(p \rightarrow \diamond_{\leq u} \text{mode} \notin \mathbf{FailureCondition}(p))$$

$$\begin{aligned} \text{CompleteTimeoutProperty}(p) = \square(\diamond_{\leq u} (\mathbf{TimeoutCondition}(p) \rightarrow \\ p \vee \mathbf{TimeoutReset}(p))) \end{aligned}$$

$$\begin{aligned} \text{CorrectTimeoutProperty}(p) = \square(\mathbf{TimeoutCondition}(p) \wedge \\ \text{O}_{\leq u} \mathbf{TimeoutReset}(p) \rightarrow \neg p) \end{aligned}$$

$$\text{FunctionProperty}(p) = \square(p = \mathbf{Function}(p))$$

$$\text{FunctionProperty}(p) = \square(p \rightarrow \text{data}(p) = \mathbf{Function}(p))$$

$$\text{InvariantProperty}(p) = \square(p \in \mathbf{InvariantRange}(p))$$

$$\text{InvariantProperty}(p) = \square(p \rightarrow \text{data}(p) \in \mathbf{InvariantRange}(p))$$

$$\begin{aligned} \text{ReactionProperty}(p) = \square((p \wedge \text{mode} \in \mathbf{ReactionCondition}(p)) \rightarrow \\ \diamond_{\in I} \mathbf{Reaction}(p)) \end{aligned}$$

$$\text{PrecededByProperty}(p) = p \rightarrow \text{O}_{\in I} \mathbf{PrecededBy}(p)$$

$$\begin{aligned} \text{PeriodProperty}(p) = (\diamond_{\leq v} \neg \text{enabled} \vee \triangleright_{[v, v+j]} p) \wedge \\ \square(\text{rise}(\text{enabled}) \rightarrow \\ (\diamond_{\leq v} \neg \text{enabled} \vee \triangleright_{[v, v+j]} p)) \wedge \\ \square((p \wedge \text{enabled}) \rightarrow \\ (\diamond_{\leq u} \neg \text{enabled} \vee \triangleright_{[u, u+j]} p)) \end{aligned}$$

$$\begin{aligned} \text{ThroughputRatio}(p) = \mathbf{PeriodInterval}(p) = \mathbf{ThroughputRatio}(p) * \\ \mathbf{PeriodInterval}(\mathbf{ThroughputInput}(p)) \end{aligned}$$

$$\text{ToleranceProperty}(p) = \square(\# p \leq \mathbf{Tolerance}(p))$$

$\triangleright_{[a,b]}\varphi$  is true when the next time  $\varphi$  is true is within  $[a, b]$  time units;  $\!|\triangleright_{[a,b]}\varphi$  is the strict version of  $\triangleright_{[a,b]}\varphi$  that does not consider the occurrence of  $\varphi$  at the current time;

Additional probabilistic properties are available, which make use of the ET and LRA operators also described in section A.6.

$$\begin{aligned} \text{MTTF}(x) &= \text{ET}(\text{mode} \in \mathbf{FailureCondition}(x)) \\ \text{MTTR}(x) &= \text{ET}(\text{mode} \notin \mathbf{FailureCondition}(x)), \\ &\quad \text{with initial state } \text{mode} \in \mathbf{FailureCondition}(x) \\ \text{Availability}(s) &= \text{LRA}(\text{mode} \notin \mathbf{FailureCondition}(s)) \end{aligned}$$

#### C.4 DESIGN ATTRIBUTE MAPPING

This section provides the complete mapping of design attributes (and the requirement types they can be found in) to CSSP properties or modeling approaches.

For each type of requirement that can be found in the taxonomy presented in Section 3.2, a list of associated design attributes is given, as well as an indicator how this requirement could be formalized. Ideally, formalization is possible by using the CSSP properties, but another approach may have to be taken instead.

The list is not intended to cover all possible types of requirements, possible attributes or even possible formalization approaches. It is limited to what is presented in Section 3.4, which is intended to provide a template from which more types and formalizations can be derived. This is discussed in more detail in that section.

(List starts on the next page)

Requirement Type	Attribute(s)	Formalization according to CSSP
Context	allocated process and memory units processing capacity, clock frequency, endianness, memory sizes, addressable memory units	Components of <b>processor</b> or <b>memory</b> type. Using <b>InvariantRange</b> or <b>Function</b> properties.
Configuration	list of sub-systems, type of sub-systems, connections between sub-systems redundancy scheme, memory protection mechanism	Subcomponents and port connections in the model.  Redundant component specifications in the model.
Physical Interface	area, volume, alignment, stiffness, tolerance, geomtry, flatness, fixation, mass and interation	Using <b>Function</b> properties to define acceptable ranges.
Electrical Interface	voltage, current, margins for electrically induced effects.	Same as above.
Thermal Interface	temperature, thermal resistance	Same as above.
Software Interface Input/Output	events, functions response to inputs, reaction time  input required to generate an output	Event ports in the model. Using <b>Reaction</b> , <b>ReactionMinDelay</b> and <b>ReactionMaxDelay</b> properties. Using <b>PrecededBy</b> properties.
Mode	modes, mode transitions, transition triggers mode invariants	Modes, states and transition directly specified in the model.  Using <b>ModeInvariant</b> properties.
Data-Handling	input data rates, output data rates persistent data processing steps	Using the <b>PeriodInterval</b> property. Using <b>Change</b> properties. Specified as modes and transitions in the model.

Monitoring	<p>output data generation monitored parameters, monitor range check frequency</p>	<p>Specified as (event) data ports in the model. Using the <b>MonitorRange</b> property, specified on ports that represent the parameters. Using the <b>MonitorDelay</b> property.</p>
	<p>parameter check response actions parameter check response result monitoring state</p>	<p>Using the <b>MonitorResponse</b> property. Using the <b>Function</b> property on the response port. Using the <b>MonitoringEnabled</b> property.</p>
Operational	report format	Specifies as the type of a data port in the model.
	data frequency	Using the <b>PeriodInterval</b> property.
	event data	Modeled as the type of an event data port.
	observation mode	Using the <b>PeriodEnabled</b> property.
	list of commands	Specified using data component types.
	type of commands	Specifies as the type of a data port in the model.
	command conditions	Using the <b>PeriodEnabled</b> property.
	command responses message types	Using <b>Response</b> properties. Specified as an implementation of the data component representing the message.
	message formats	Same as above.
	message rates, jitter	Using <b>PeriodInterval</b> and <b>PeriodJitter</b> properties.
Performance	communication windows	Using the <b>PeriodEnabled</b> property.
	latency, response times, deadlines throughput	Using <b>ReactionMinDelay</b> and <b>ReactionMaxDelay</b> properties. Specified by <b>ThroughputInput</b> and <b>ThroughputRation</b> properties.

	CPU load, processing capacity, memory capacity, communication capacity	Either specified by components in the model directly, or monitored using the <b>MonitorRange</b> property.
Dependability	kind of failure	Using the <b>FailureCondition</b> property to specify the failure configuration.
	failure detection delay	Using <b>AlarmDelay</b> .
	recovery actions	Using <b>Reaction</b> properties to indicate the recover event.
Reliability, Maintainability	mean time to failure (MTTF), mean time to repair (MTTR)	Using the <b>FailureCondition</b> property to specify failed states.
Availability	availability	Same as above.
Dependability, Safety	failure tolerance	Using <b>Tolerance</b> properties on event ports representing errors, or <b>InvariantRange</b> properties on input data.

---

# D

## OUTPUT/ARTIFACTS

This appendix provides an overview of the various artifacts that COMPASS may produce as a result of any of its analyses. This includes some of their theoretical foundations and examples, though the provided references contain more in-depth information.

### D.1 TRACES

One of the verification outputs of COMPASS is the trace. A trace provides a sequence of states in the model that lead up to a particular state of interest, for instance when a counterexample has been found. A trace in COMPASS contains the evolution of each variable in the model as a separate row, with each column representing a single state in the trace. An example is shown in Figure D.1.

Traces provide both counterexamples and witnesses for certain properties. For counterexamples, generally the last step of the trace provides the state which invalidates some property of interest. Both for counterexamples and witnesses, the trace may contain a loop that maps it to an infinite path. Such loops start at a given step (indicated in the GUI with a chevron) and continue until the last step.

Boolean values, as well as the occurrence of events, are indicated with either a low (false) or high (true) line. Other values are shown textually in between lines, which will cross from high to low and vice versa between steps when the value changes.

### D.2 FAULT TREES

Fault trees (FTs), and the associated fault tree analysis (FTA), are a well established method in reliability engineering. They are tree structured graphs that describe how faults can propagate through the system, and how their interaction may trigger further faults. At the top of the tree a particular system failure is considered. FTA

Name	Step1	Step2	Step3	Step4	Step5	Step6	Step7	Step8
done_delta								
event_#tau								
event_position								
root.active								
root.data_#fix								
root.data_#position	◊ (0)	◊ (0)	◊ (0)	◊ (0)	◊ (0)	◊ (2)	◊ (4)	◊ (2)
root.event_#tau								
root.event_position								
root.id	◊ (1)	◊ (2)	◊ (1)	◊ (1)	◊ (8)	◊ (3)	◊ (1)	◊ (2)
root.mode	◊ de_m_noi	◊ de_m_noi	◊ ode_m_noi	◊ ode_m_noi	◊ ode_m_noi	◊ mode_m_s	◊ mode_m_s	◊ mode_m_s
root.resolved_position								
root.sc_has_fix_prim								
root.sc_has_fix_sec								
root.sc_primary.active								

FIGURE D.1 – Example trace as shown by the COMPASS toolset.

concerns itself with determining how and with what probability such a failure may occur [130, 136].

A fault tree consists of events which represent the faults that may occur (either directly or triggered by some other condition), and gates that dictate how combinations of faults may propagate further up in the tree. In COMPASS, the considered events are *basic events*, which lie at the bottom of the tree (leaves in the graph) and are associated with a fixed error rate which determine the likelihood of the event occurring in a certain time period, based on the exponential distribution. The gates are either AND- or OR-gates. AND-gates trigger when all of their children have triggered. Or-gates trigger when at least one child has triggered.

A particular fault tree represents the possible way a particular failure in or of the system may occur. The root node of the tree is also called the *top level event*, which is the failure which is considered by the fault tree. An example is given in Figure D.2.

To determine the probability of failure — that is, the probability that the top level event is triggered — the probability of each child failing is determined recursively, and those are then aggregated according to the type of the parent gate. For AND-gates, the probabilities can be multiplied, whereas for OR-gates they can be summed, minus the probability they occur simultaneously. For example, given an AND-gate  $A$  and two children  $E_1$  and  $E_2$ , it follows that  $\Pr[A] = \Pr[E_1 \cap E_2] = \Pr[E_1] * \Pr[E_2]$ . For basic events associated with a failure rate  $\lambda$ , the probability of failure within some time bound  $t$  is given by  $1 - e^{-\lambda t}$ .

### D.2.1 DYNAMIC FAULT TREES

Classically, FTs consider static AND- and OR-gates, which trigger when respectively all children or a single child is triggered. Later extensions introduced different gates, in particular those that depend on a particular *ordering* of child events. These

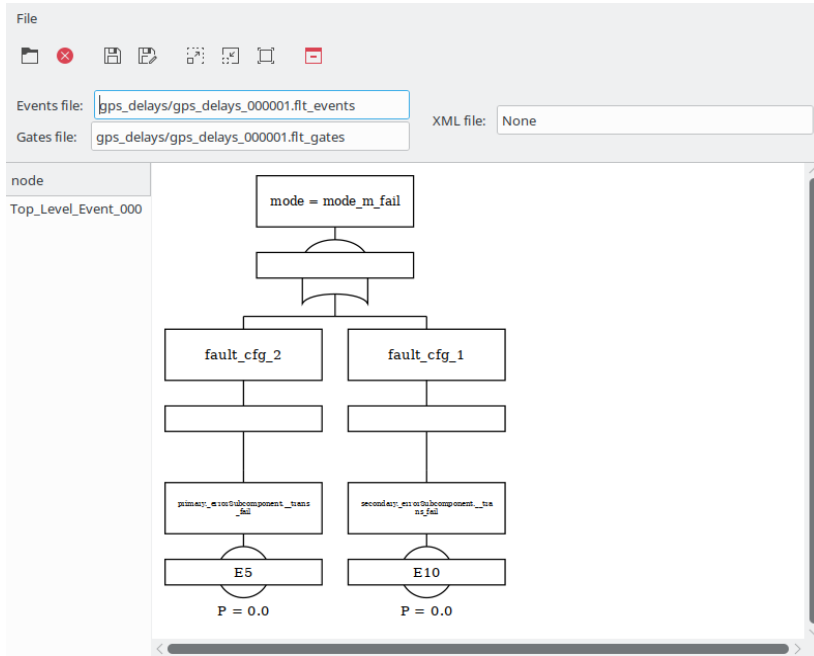


FIGURE D.2 – Example fault tree as generated and displayed by COMPASS. The top level event `mode = mode:fail` is triggered by either of the two basic events triggered by `trans_fail`.

are the dynamic fault trees (DFTs). Such orderings may impose restrictions on the possible failure configurations of the system, or preventing faults from propagating (meaning a failed-safe state has been reached).

In COMPASS, the only dynamic gate considered is the *priority AND* (PAND)-gate, which triggers only if all of its children fail in a fixed sequence (graphically, from left to right), and enters a failed-safe state if at least one child fails in a different order. Other gates exist as well, in particular gates that make the DFT a directed acyclic graph (DAG) instead of a tree. Examples are *spare*-gates, which can share child nodes as a pool of spares that may be used in case of failure, and *functional dependency* (FDEP)-gates, where the failure of a specially designated child will directly trigger the failure of all other children as if it were the cause of a dependent failure.

Unlike FTs, where static probabilities may be considered, for DFTs time must be taken into account as well, leading to the fact that they reason over (continuous) probability distributions. However, as COMPASS always considers failure rates for basic events, and not static probabilities, this is already taken into account. In particular, when performing fault tree verification (cf. Section 5.3.6) the corresponding cumulative distribution function (CDF) is presented to the user.

The semantics of DFTs are still a topic of research, and care should be taken which are applied [84]. As COMPASS limits itself to the use of PAND-gates and a fairly rigid structure based on minimal cut-sets (MCSs), the semantics remain clear. However, this does not hold in general, depending on the structure of the DFT and the gates used.

### D.3 TIMED FAILURE PROPAGATION GRAPHS

Timed failure propagation graphs (TFPGs) [23, 106] are a graphical representation of how a fault, or failure, can propagate through the system, in particular with respect to the delays between occurrence of a fault and a subsequent propagation. Their structure is similar to that of fault trees, but they consider different aspects of safety and reliability analysis, in particular with regard to timing.

A TFPG consists of *failure modes* and *discrepancy nodes*, with edges in between them that specify the possible propagation directions of faults. In a TFPG, a fault originates from one of the specified failure modes and propagates over the edges to discrepancy nodes. The discrepancy nodes dictate under what conditions a fault may propagate, distinguishing between AND- and OR-nodes.

Unlike a FT, a TFPG permits the modes of a system to be considered as part of the state, with connections between failure modes and discrepancies to be dependent on this mode. This means that a change of the system mode can prevent or allow certain propagation to occur (though fault trees can use conditioning events to model such behavior). Every edge in a TFPG is annotated with the system modes in which it is enabled.

As the name implies, a TFPG also associates time delays with the possible propagations. These are specified as ranges, with a lower bound from  $\mathbb{R}_{\geq 0}$  and an upper bound from  $\mathbb{R}_{\geq 0} \cup \{\infty\}$ . When a failure mode or discrepancy node is triggered, a propagation may move to another node based on a time delay from the given interval.

Two types of discrepancy nodes are considered: *AND* and *OR* nodes. The prior will only propagate a fault once all incoming edges have triggered a propagation. The latter will do so if at least one edge has been triggered (these semantics are similar to that of fault trees).

Discrepancy nodes can be considered monitored or unmonitored. A monitored discrepancy node has an associated *alarm*, that is triggered when the discrepancy is triggered. This information may be used for diagnosability analysis, or fault isolation.

Formally, a TFPG is a graph

$$t = (F, D, E, M, ET, EM) ,$$

where

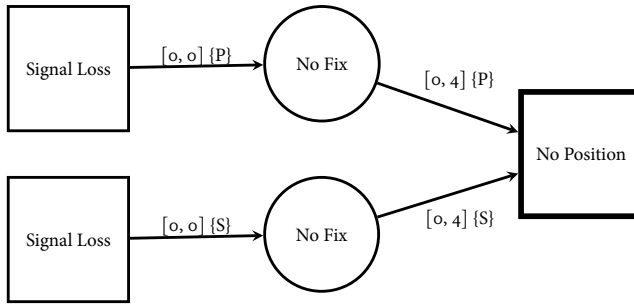


FIGURE D.3 – Example TFPG, which models the propagation of signal loss of two redundant GPS units. After losing a signal, a unit immediately loses its fix, which reports loss of position after 4 time units. When both units fail, the location can no longer be determined.

- »  $F$  is the set of *failure* modes;
- »  $D$  is the set of discrepancies, where  $D = D_{\text{AND}} \cup D_{\text{OR}}$ , the union of AND and OR discrepancies;
- »  $E$  is the set of edges where  $E \subseteq V \times V$ , with  $V = F \cup D$ ;
- »  $M$  is the set of system modes;
- »  $\text{ET} : E \rightarrow [\mathbb{R}_{\geq 0}, \mathbb{R}_{\geq 0} \cup \{\infty\}]$  is the map from edges to time delays  $[t_{\min}, t_{\max}]$ , with  $t_{\min} \leq t_{\max}$ ;
- »  $\text{EM} : E \rightarrow 2^M$  is the map from edges to system modes;

Graphically, a TFPG can be depicted as in the example from Figure D.3. Failure modes are drawn as dotted squares. Discrepancy nodes are drawn with a solid outline, either as a square for AND-nodes or a circle for OR-nodes. In COMPASS, discrepancies may be monitored, graphically depicted by a thick outline. These correspond to observable expressions in SLIM.

Upon occurrence of a fault in one of the failure modes, it may propagate along any of its outgoing edges  $e \in E$ , with a delay  $t \in \text{ET}(e)$ , if that edge is enabled. Edges are enabled if the edge is labeled with the current system mode  $m \in M$ , i.e.,  $m \in \text{EM}(e)$ .

A TFPG does not maintain memory for its edges, meaning a propagation is independent of any previous propagations. Furthermore, when an edge gets deactivated due to a change of the system mode, the propagation is “stopped”.



# ACRONYMS

A	AADL	Architecture Analysis & Design Language
	ACTL	action-based CTL
	ADC	analog to digital converter
	ADCS	attitude determination and control system
	ADD	algebraic decision diagram
	ANTLR	another tool for language recognition
	AP	atomic proposition
	ASM	abstract state machine
	ASSERT	automated proof based system and software engineering for real-time applications
	AUTOGEF	Automated Model Generation Toolset for FDIR
B	BDD	binary decision diagram
	BIP	behavior, interaction, priority
	BMC	bounded model checking
C	CATSY	Catalogue of System and Software Properties
	CBA	contract-based analysis
	CBD	contract-based design
	CDF	cumulative distribution function
	CDMS	command and data management system
	CDR	critical design review
	CGM	COMPASS Graphical Modeler
	CH	Chernoff-Hoeffding
	CI	confidence interval
	CLI	command line interface
	CLT	central limit theorem
	CNF	conjunctive normal form
	CNL	controlled natural language
	COM	communication system
	COMPASS	Correctness, Modeling and Performance of Aerospace Systems
	CORDET	Component Oriented Development Techniques
	COTS	commercial off-the-shelf
	CPU	central processing unit
	CR	Chow-Robbins

	CRC	cyclic redundancy check
	CSL	continuous stochastic logic
	CSSP	catalogue of system and software properties
	CTL	computation tree logic
	CTMC	continuous-time Markov chain
D	D-MILS	distributed MILS
	DAG	directed acyclic graph
	DD	decision diagram
	DFT	dynamic fault tree
	DPU	data processing unit
	DTMC	discrete-time Markov chain
E	ECSS	European Cooperation for Space Standardization
	EDA	event data automaton
	EMA	error modeling annex
	EMF	Eclipse Modeling Framework
	EPS	electrical power system
	ESA	European Space Agency
	ET	expected time
	EU	European Union
F	FAME	failure and anomaly management engineering
	FDIR	failure detection, isolation and recovery
	FMEA	failure modes and effects analysis
	FMECA	failure mode, effects and criticality analysis
	FSAP	Formal Safety Assessment Platform
	FSM	finite-state model
	FT	fault tree
	FT	Formal Tropos
	FTA	fault tree analysis
G	GIS	geographic information system
	GLONASS	global navigation satellite system
	GNSS	global navigation satellite system
	GPIO	general purpose IO
	GPS	global positioning system
	GSMP	generalized semi-Markov process
	GUI	graphical user interface

H	HASDEL	Hardware Software Dependability for Launchers
	HK	housekeeping
I	I <sup>2</sup> C	inter-integrated circuit
	IC3	incremental construction of inductive clauses for indubitable correctness
	IMC	interactive Markov chain
	IMCA	interactive Markov chain analyzer
	IO	input/output
K	KAOS	knowledge acquisition in automated specification
L	LRA	long-run average
	LTL	linear temporal logic
	LTS	labeled transition system
M	MBSSE	model-based system and software engineering
	MCS	minimal cut-set
	MDE	model-driven engineering
	MDP	Markov decision process
	MEC	maximal end component
	MILS	multiple independent levels of security
	MITL	metric interval temporal logic
	MRM	Markov reward model
	MRMC	Markov reward model checker
	MSC	message sequence chart
	MSO	monadic second order
	MTBDD	multi terminal BDD
	MTBF	mean time between failures
	MTL	metric temporal logic
	MTTF	mean time to failure
	MTTFF	mean time to first failure
	MTTR	mean time to repair
N	NBA	non-deterministic Büchi automaton
	NEDA	network of event data automata
	NER	named entity recognition
	NFR	non-functional requirement
	NLP	natural language processing

O	OBC	on-board computer
	OCRA	Othello Contracts Refinement Analysis
	ODE	ordinary differential equation
	OPC	Open Packaging Conventions
	OSRA	On-board Software Reference Architecture
	OSS	OCRA system specification
P	PCDU	power conditioning and distribution unit
	PCTL	probabilistic computation tree logic
	PDF	probability distribution function
	PDR	preliminary design review
	PL	Payload
	PLTL	probabilistic linear temporal logic
	PRNG	pseudo random number generator
	PSL	property specification language
	PTA	priced timed automaton
R	RAM	random access memory
	RAMS	reliability, availability, maintainability and safety
	RNG	random number generator
	RSS	resident set size
S	SAT	satisfiability
	SAVOIR	Space Avionics Open Interface Architecture
	SBMC	simple bounded model checking
	SCR	software cost reduction
	SEU	single event upset
	SFMT	SIMD-oriented Fast Mersenne Twister
	SIMD	single instruction multiple data
	SLIM	System Level Integration Modeling
	SMC	statistical model checking
	SMV	symbolic model verifier
	SPN	stochastic Petri net
	SPRT	sequential probability ratio test
	SSP	single sample plan
	SST	simple sequential test
	STA	stochastic timed automaton
T	TASTE	The ASSERT Set of Tools for Engineering
	TFFPG	timed failure propagation graph

TFPM    timed failure propagation model  
TLE      top-level event  
TTC      telemetry, tracking and control

U    USB      Universal Serial Bus

X    XMI      XML metadata interchange  
      XML      extensible markup language  
      xSAP     eXtended Safety Assessment Platform



# GLOSSARY

- alarm**  
Event that signals a fault or failure, generally as part of an FDIR system.
- atomic proposition**  
A property specified in propositional logic without logical connectives, referring to expressions over state variables.
- bounded temporal property**  
A temporal property that reasons over events up to some finite horizon.
- clock**  
A variable that changes value proportionally to the progress of time, and can be reset to a constant value.
- counterexample**  
A formal artifact that disproves a property for a given model.
- criticality**  
The degree of importance.
- discrete model**  
A model of which the variables change only in discrete steps.
- error**  
An inaccuracy or incorrect input/behavior
- explicit**  
Represented directly.
- failed-safe**  
State in which a fault is no longer capable of propagating.
- failure**  
The inability to perform a specified or desired function.
- fairness**  
The absence of unrealistic infinite behavior.
- fault**  
A deviation from specified behavior due to an error.
- formal property**  
A formula expressed in a formal logic. Formal properties have well-defined semantics and can be mathematically proven.
- hybrid model**  
A model of which the variables change in discrete steps or according to differential equations.
- I<sup>2</sup>C**  
A simple, low-cost serial bus, supporting multi-master, multi-slave configurations.
- launcher**  
A vehicle that can bring a payload to high altitude or into orbit around Earth.
- MEC**  
Maximal connected sub-automaton, of which all states are pairwise reachable, and is not subsumed by another MEC.

- mode**  
A particular configuration of a system and its possible subsystems.
- model**  
A (simplified) representation of a system with formally defined semantics expressed in states and transitions.
- model checking**  
Verification of desired behavioral properties by systematically exploring the reachable states of a model.
- mutex**  
A synchronization mechanism for enforcing limits on mutual access to a particular resource.
- non-determinism**  
An abstract notion of underspecification, where a single action leads to multiple outcomes.
- path**  
A sequence of states.
- probabilistic**  
Predictable according to some probability distribution.
- propositional property**  
formula expressed in a propositional logic.
- random**  
Of unpredictable nature.
- reactive system**  
A system that performs its tasks in response to external stimuli.
- redundancy**  
The presence of a system with duplicate functionality to improve reliability.
- redundancy**  
The presence of a system with duplicate functionality to improve reliability.
- reliability**  
The (expected) time of continuous failure-free operation of a system.
- robustness**  
The degree to which a system can tolerate and recover from failures.
- safe mode**  
Operational mode of a system with the primary focus of ensuring the safety of the system.
- satellite**  
An object that is in orbit. Usually refers to a man-made device, most often used for communications or observation.
- specification**  
An exact description of a concept, from which instances can be derived, or be compared against.
- state**  
A particular assignment of values in a model or system.
- state space**  
The set or subset of states described by a model.
- state space explosion**  
The phenomenon where a small increase in the size of a model results in (exponential) blowup of the state space size.

- symbolic**  
Represented by a (formal) description.
- system property**  
A particular aspect of a component in a system, such as reliability or timing.
- technology readiness level**  
Technical maturity of a specific space application.
- temporal property**  
A formula expressed in a temporal logic, such as LTL or CTL. Generally designated using  $\varphi$  or  $\psi$ .
- timed model**  
A model of which the variables change in discrete steps or linear according to the progress of time.
- trace**  
See path.
- transition**  
A step from one state to another.
- validation**  
The process of assuring a system fulfills the needs and expectations of stakeholders.
- verification**  
The process of ensuring a system meets all design criteria.
- wizard**  
A sequence of dialogs that guides a user step-wise through a (complex) task.



# BIBLIOGRAPHY

- [1] *AADL Inspector*. URL: <https://www.ellidiss.com/products/aadl-inspector/> (cited on page 99).
- [2] J.-R. Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005. ISBN: 978-0-521-02175-3 (cited on page 53).
- [3] E. Alana et al. “Automated generation of FDIR for the COMPASS integrated toolset (AUTOGEF)”. In: *DASIA 2012 – DAta Systems In Aerospace*. Proceedings. Ed. by L. Ouwehand. Vol. SP-701. Drubrovnik, Croatia: ESA Communications, 2012. ISBN: 978-92-9092-265-0 (cited on page 92).
- [4] M. W. Alford et al., eds. *Distributed Systems: Methods and Tools for Specification, An Advanced Course*. (). Vol. 190. LNCS. Munich, Germany: Springer, 1985. ISBN: 3-540-15216-4. DOI: 10.1007/3-540-15216-4 (cited on page 53).
- [5] B. Alpern and F. B. Schneider. “Recognizing safety and liveness”. In: *Distributed Computing* 2.3 (1987), pp. 117–126. DOI: 10.1007/BF01782772 (cited on page 53).
- [6] R. Alur, T. Feder, and T. A. Henzinger. “The benefits of relaxing punctuality”. In: *Journal of the ACM* 43.1 (1996), pp. 116–146. DOI: 10.1145/227595.227602 (cited on page 52).
- [7] R. Alur and T. A. Henzinger, eds. *8th International Conference on Computer Aided Verification, CAV '96*. Proceedings. (July 31–Aug. 3, 1996). Vol. 1102. LNCS. New Brunswick, NJ, USA: Springer, 1996. ISBN: 3-540-61474-5. DOI: 10.1007/3-540-61474-5.
- [8] V. Ambriola and V. Gervasi. “On the systematic analysis of natural language requirements with CIRCE”. In: *Automated Software Engineering* 13.1 (2006), pp. 107–167. DOI: 10.1007/s10515-006-5468-2 (cited on page 56).
- [9] L. Armbrorst. “Generating Simulink Models from AADL system descriptions”. Bachelor’s Thesis. RWTH Aachen University, Oct. 2015 (cited on pages 6, 125).
- [10] M. Autili et al. “Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar”. In: *IEEE Transactions on Software Engineering* 41.7 (2015), pp. 620–638. DOI: 10.1109/TSE.2015.2398877 (cited on pages 53, 55).
- [11] *AutoFOCUS3*. URL: <https://af3.fortiss.org/> (cited on page 122).

- [12] A. Aziz et al. “Verifying continuous time Markov chains”. In: *8th International Conference on Computer Aided Verification, CAV ’96*. Proceedings. (July 31–Aug. 3, 1996). Ed. by R. Alur and T. A. Henzinger. Vol. 1102. LNCS. New Brunswick, NJ, USA: Springer, 1996, pp. 269–276. ISBN: 3-540-61474-5. DOI: 10.1007/3-540-61474-5\_75 (cited on pages 52, 191).
- [13] C. Baier et al. “Model-checking algorithms for continuous-time Markov chains”. In: *IEEE Transactions on Software Engineering* 29.6 (2003), pp. 524–541. DOI: 10.1109/TSE.2003.1205180 (cited on page 52).
- [14] P. Ballarini et al. “COSMOS: a statistical model checker for the hybrid automata stochastic logic”. In: *8th International Conference on Quantitative Evaluation of Systems, QEST 2011*. (Sept. 5–8, 2011). Aachen, Germany: IEEE Computer Society, 2011, pp. 143–144. ISBN: 978-1-4577-0973-9. DOI: 10.1109/QEST.2011.24 (cited on page 69).
- [15] A. Basu et al. “Rigorous component-based system design using the BIP framework”. In: *IEEE Software* 28.3 (2011), pp. 41–48. DOI: 10.1109/MS.2011.27 (cited on page 145).
- [16] P. Bellini, P. Nesi, and D. Rogai. “Expressing and organizing real-time specification patterns via temporal logics”. In: *Journal of Systems and Software* 82.2 (2009), pp. 183–196. DOI: 10.1016/j.jss.2008.06.041 (cited on page 53).
- [17] S. Bensalem et al. “D-Finder: a tool for compositional deadlock detection and verification”. In: *21st International Conference on Computer Aided Verification, CAV 2009*. Proceedings. (June 26–July 2, 2009). Ed. by A. Bouajjani and O. Maler. Vol. 5643. LNCS. Grenoble, France: Springer, 2009, pp. 614–619. ISBN: 978-3-642-02657-7. DOI: 10.1007/978-3-642-02658-4\_45 (cited on page 183).
- [18] S. Berezin, S. V. A. Campos, and E. M. Clarke. “Compositional reasoning in model checking”. In: *Compositionality: The Significant Difference, International Symposium, COMPOS’97*. Revised Lectures. (Sept. 8–12, 1997). Ed. by W. P. de Roever, H. Langmaack, and A. Pnueli. Vol. 1536. LNCS. Bad Malente, Germany: Springer, 1997, pp. 81–102. ISBN: 3-540-65493-3. DOI: 10.1007/3-540-49213-5\_4 (cited on page 109).
- [19] A. Biere and R. Bloem, eds. *26th International Conference on Computer Aided Verification, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014*. Proceedings. Vol. 8559. LNCS. Vienna, Austria: Springer, 2014. ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9.
- [20] *BIP Compiler*. URL: <https://www-verimag.imag.fr/New-BIP-tools.html> (cited on page 144).
- [21] *BIP2 Documentation*. URL: <https://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/pdf/BIP2.pdf> (cited on pages 144, 145).
- [22] *BIP-to-NuSMV transformation tool*. URL: <https://risd.epfl.ch/bip2nusmv> (cited on page 183).

- [23] B. Bittner, M. Bozzano, and A. Cimatti. “Automated synthesis of timed failure propagation graphs”. In: *25th International Joint Conference on Artificial Intelligence, IJCAI 2016*. Proceedings. (July 9–15, 2016). Ed. by S. Kambhampati. New York, NY, USA: IJCAI/AAAI Press, 2016, pp. 972–978. ISBN: 978-1-57735-770-4 (cited on pages 119, 120, 216).
- [24] B. Bittner et al. “Automated verification and tightening of failure propagation models”. In: *30th AAAI Conference on Artificial Intelligence*. Proceedings. (Feb. 12–17, 2016). Ed. by D. Schuurmans and M. P. Wellman. Phoenix, Arizona, USA: AAAI Press, 2016, pp. 907–913. ISBN: 978-1-57735-760-5. URL: <https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12037> (cited on page 121).
- [25] B. Bittner et al. “The xsap safety analysis platform”. In: *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016*. Proceedings. (Apr. 2–8, 2016). Ed. by M. Chechik and J.-F. Raskin. Vol. 9636. LNCS. Eindhoven, The Netherlands: Springer, 2016, pp. 533–539. ISBN: 978-3-662-49673-2. DOI: 10.1007/978-3-662-49674-9\_31 (cited on page 114).
- [26] S. Bliudze and J. Sifakis. “The algebra of connectors - structuring interaction in BIP”. In: *IEEE Transactions on Computers* 57.10 (2008), pp. 1315–1330. DOI: 10.1109/TC.2008.26 (cited on page 145).
- [27] J. Bogdoll, A. Hartmanns, and H. Hermanns. “Simulation and statistical model checking for modestly nondeterministic models”. In: *16th International GI/ITG Conference on Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance, MMB & DFT 2012*. Proceedings. (Mar. 19–21, 2012). Ed. by J. B. Schmitt. Vol. 7201. LNCS. Kaiserslautern, Germany: Springer, 2012, pp. 249–252. ISBN: 978-3-642-28539-4. DOI: 10.1007/978-3-642-28540-0\_20 (cited on pages 69, 71).
- [28] A. Bouajjani et al. “Safety for branching time semantics”. In: *18th International Colloquium on Automata, Languages and Programming, ICALP 91*. Proceedings. (July 8–12, 1991). Ed. by J. L. Albert, B. Monien, and M. Rodríguez-Artalejo. Vol. 510. LNCS. Madrid, Spain: Springer, 1991, pp. 76–92. ISBN: 3-540-54233-7. DOI: 10.1007/3-540-54233-7\_126 (cited on page 53).
- [29] H. Boudali, P. Crouzen, and M. Stoelinga. “A compositional semantics for dynamic fault trees in terms of interactive Markov chains”. In: *5th International Symposium on Automated Technology for Verification and Analysis, ATVA 2007*. Proceedings. (Oct. 22–25, 2007). Ed. by K. S. Namjoshi et al. Vol. 4762. LNCS. Tokyo, Japan: Springer, 2007, pp. 441–456. ISBN: 978-3-540-75595-1. DOI: 10.1007/978-3-540-75596-8\_31 (cited on page 98).
- [30] H. Bowman, R. Gómez, and L. Su. “A tool for the syntactic detection of Zeno-timelocks in timed automata”. In: *Electronic Notes in Theoretical Computer Science* 139.1 (2005), pp. 25–47. DOI: 10.1016/j.entcs.2005.09.006 (cited on page 75).

- [31] B. Boyer et al. “PLASMA-lab: a flexible, distributable statistical model checking library”. In: *10th International Conference on Quantitative Evaluation of Systems, QEST 2013*. Proceedings. (Aug. 27–30, 2013). Ed. by K. R. Joshi et al. Vol. 8054. LNCS. Buenos Aires, Argentina: Springer, 2013, pp. 160–164. ISBN: 978-3-642-40195-4. DOI: 10.1007/978-3-642-40196-1\_12 (cited on page 69).
- [32] M. Bozzano, A. Cimatti, and F. Tapparo. “Symbolic fault tree analysis for reactive systems”. In: *5th International Symposium on Automated Technology for Verification and Analysis, ATVA 2007*. Proceedings. (Oct. 22–25, 2007). Ed. by K. S. Namjoshi et al. Vol. 4762. LNCS. Tokyo, Japan: Springer, 2007, pp. 162–176. ISBN: 978-3-540-75595-1. DOI: 10.1007/978-3-540-75596-8\_13 (cited on page 114).
- [33] M. Bozzano et al. “Formal design of fault detection and identification components using temporal epistemic logic”. In: *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*. Proceedings. (Apr. 5–13, 2014). Ed. by E. Ábrahám and K. Havelund. Vol. 8413. LNCS. Grenoble, France: Springer, 2014, pp. 326–340. ISBN: 978-3-642-54861-1. DOI: 10.1007/978-3-642-54862-8\_22 (cited on pages 53, 123).
- [34] M. Bozzano et al. “Formal safety assessment via contract-based design”. In: *12th International Symposium on Automated Technology for Verification and Analysis, ATVA 2014*. Proceedings. (Nov. 3–7, 2014). Ed. by F. Cassez and J.-F. Raskin. Vol. 8837. LNCS. Sydney, NSW, Australia: Springer, 2014, pp. 81–97. ISBN: 978-3-319-11935-9. DOI: 10.1007/978-3-319-11936-6\_7 (cited on page 115).
- [35] M. Bozzano et al. “Formal design of asynchronous fault detection and identification components using temporal epistemic logic”. In: *Logical Methods in Computer Science* 11.4 (2015). DOI: 10.2168/LMCS-11(4:4)2015 (cited on page 123).
- [36] M. Bozzano et al. *COMPASS 3 Deliverable D4: Roadmap*. Project Deliverable. FBK and RWTH Aachen, 2016 (cited on page 122).
- [37] C. E. Budde, P. R. D’Argenio, and H. Hermanns. “Rare event simulation with fully automated importance splitting”. In: *12th European Workshop on Computer Performance Engineering, EPEW 2015*. Proceedings. (Aug. 31–Sept. 1, 2015). Ed. by M. Beltrán, W. J. Knottenbelt, and J. T. Bradley. Vol. 9272. LNCS. Madrid, Spain: Springer, 2015, pp. 275–290. ISBN: 978-3-319-23266-9. DOI: 10.1007/978-3-319-23267-6\_18 (cited on page 76).
- [38] R. Cavada et al. “The nuxmv symbolic model checker”. In: *26th International Conference on Computer Aided Verification, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014*. Proceedings. Ed. by A. Biere and R. Bloem. Vol. 8559. LNCS. Vienna, Austria: Springer, 2014, pp. 334–342.

ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9\_22 (cited on page 96).

- [39] T. Chen et al. “Model checking of continuous-time Markov chains against timed automata specifications”. In: *Logical Methods in Computer Science* 7.1 (2011). DOI: 10.2168/LMCS-7(1:12)2011 (cited on page 52).
- [40] B. H. Cheng et al. “Using security patterns to model and analyze security requirements”. In: *Workshop on Requirements for High-Assurance Systems, RHAS03*. Proceedings. Monterey Bay, CA, USA, Sept. 2003, pp. 13–22 (cited on page 53).
- [41] A. Chin et al. “CubeSat: the pico-satellite standard for research and education”. In: *AIAA Space 2008 Conference & Exposition*. 2008, p. 7734 (cited on page 143).
- [42] L. Chung et al. *Non-functional requirements in software engineering*. Vol. 5. Springer Science & Business Media, 2012 (cited on page 47).
- [43] A. Cimatti, R. Demasi, and S. Tonetta. “Tightening the contract refinements of a system architecture”. In: *Formal Methods in System Design* 52.1 (2018), pp. 88–116. DOI: 10.1007/s10703-017-0312-9 (cited on page 111).
- [44] A. Cimatti, M. Dorigatti, and S. Tonetta. “OCRA: a tool for checking the refinement of temporal contracts”. In: *28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*. (Nov. 11–15, 2013). Ed. by E. Denney, T. Bultan, and A. Zeller. Silicon Valley, CA, USA: IEEE, 2013, pp. 702–705. DOI: 10.1109/ASE.2013.6693137 (cited on page 96).
- [45] A. Cimatti, C. Pecheur, and R. Cavada. “Formal verification of diagnosability via symbolic model checking”. In: *18th International Joint Conference on Artificial Intelligence, IJCAI-03*. (Aug. 9–15, 2003). Ed. by G. Gottlob and T. Walsh. Acapulco, Mexico: Morgan Kaufmann, 2003, pp. 363–369. URL: <https://ijcai.org/proceedings/2003> (cited on page 118).
- [46] A. Cimatti and S. Tonetta. “Contracts-refinement proof system for component-based embedded systems”. In: *Science of Computer Programming* 97 (2015), pp. 333–348. DOI: 10.1016/j.scico.2014.06.011 (cited on page 57).
- [47] A. Cimatti et al. “NuSMV 2: an opensource tool for symbolic model checking”. In: *14th International Conference on Computer Aided Verification, CAV 2002*. Proceedings. (July 27–31, 2002). Ed. by E. Brinksma and K. G. Larsen. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, 2002, pp. 359–364. ISBN: 3-540-43997-8. DOI: 10.1007/3-540-45657-0\_29 (cited on page 96).
- [48] A. Cimatti et al. “Verifying LTL properties of hybrid systems with k-live-ness”. In: *26th International Conference on Computer Aided Verification, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014*. Proceedings. Ed. by A. Biere and R. Bloem. Vol. 8559. LNCS. Vienna, Austria: Springer, 2014, pp. 424–440. ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9\_28 (cited on page 100).

- [49] E. M. Clarke, E. A. Emerson, and A. P. Sistla. “Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach”. In: *10th Annual ACM Symposium on Principles of Programming Languages*. Conference Record. (Jan. 1983). Ed. by J. R. Wright et al. Austin, Texas, USA: ACM, 1983, pp. 117–126. ISBN: 0-89791-090-7. DOI: 10.1145/567067.567080 (cited on pages 52, 190).
- [50] *COMPASS Projects*. URL: <http://www.compass-toolset.org/projects/> (cited on pages 3, 92).
- [51] *The Future of COMPASS*. Workshop Summary Report. ESTEC, Oct. 2015 (cited on page 3).
- [52] *CORDeT*. URL: <http://cordet.gmv.com/> (cited on page 124).
- [53] A. David et al. “Statistical model checking for networks of priced timed automata”. In: *9th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS 2011*. Proceedings. (Sept. 21–23, 2011). Ed. by U. Fahrenberg and S. Tripakis. Vol. 6919. LNCS. Aalborg, Denmark: Springer, 2011, pp. 80–96. ISBN: 978-3-642-24309-7. DOI: 10.1007/978-3-642-24310-3\_7 (cited on page 69).
- [54] *D-MILS Project Website*. URL: <http://www.d-mils.org/> (cited on page 99).
- [55] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. “Property specification patterns for finite-state verification”. In: *2nd Workshop on Formal Methods in Software Practice*. Proceedings. (Mar. 4–5, 1998). Ed. by M. A. Ardis and J. M. Atlee. Clearwater Beach, Florida, USA: ACM, 1998, pp. 7–15. ISBN: 0-89791-954-8. DOI: 10.1145/298595.298598 (cited on page 53).
- [56] *European Cooperation for Space Standardization*. URL: <http://ecss.nl/> (cited on page 2).
- [57] *Ellidiss Software*. URL: <https://www.ellidiss.com/> (cited on page 99).
- [58] *Eclipse Modeling Framework*. Accessed: 2018-03-01. URL: <https://www.eclipse.org/modeling/emf/> (cited on page 98).
- [59] *System-Software Co-Engineering: Performance and Verification*. Statement of Work TRP T607-06EM. Issue 1.0, Revision 0. European Space Agency, June 2007 (cited on page 3).
- [60] M.-A. Esteve et al. “Formal correctness, safety, dependability, and performance analysis of a satellite”. In: *34th International Conference on Software Engineering, ICSE 2012*. (June 2–9, 2012). Ed. by M. Glinz, G. C. Murphy, and M. Pezzè. Zurich, Switzerland: IEEE Computer Society, 2012, pp. 1022–1031. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227118 (cited on page 144).
- [61] A. Fantechi et al. “Assisting requirement formalization by means of natural language translation”. In: *Formal Methods in System Design 4.3* (1994), pp. 243–263. DOI: 10.1007/BF01384048 (cited on page 56).

- [62] M. Gario et al. “Model checking at scale: automated air traffic control design space exploration”. In: *28th International Conference on Computer Aided Verification, CAV 2016*. Proceedings, Part II. (July 17–23, 2016). Ed. by S. Chaudhuri and A. Farzan. Vol. 9780. LNCS. Toronto, ON, Canada: Springer, 2016, pp. 3–22. ISBN: 978-3-319-41539-0. DOI: 10.1007/978-3-319-41540-6\_1 (cited on page 123).
- [63] V. Gervasi and D. Zowghi. “Reasoning about inconsistencies in natural language requirements”. In: *ACM Transactions on Software Engineering and Methodology* 14.3 (2005), pp. 277–330. DOI: 10.1145/1072997.1072999 (cited on page 56).
- [64] M. Glinz. “On non-functional requirements”. In: *15th IEEE International Requirements Engineering Conference, RE 2007*. (Oct. 15–19, 2007). New Delhi, India: IEEE Computer Society, 2007, pp. 21–26. ISBN: 0-7695-2935-6. DOI: 10.1109/RE.2007.45 (cited on page 47).
- [65] S. Gottschalk. “Modellbasiertes Testen von AADL-Modellen mit Echtzeit”. Diplomarbeit. RWTH Aachen University, 2013 (cited on page 123).
- [66] E. Grädel, W. Thomas, and T. Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research*. (Feb. 2001). Vol. 2500. LNCS. Outcome of a Dagstuhl seminar. Springer, 2002. ISBN: 3-540-00388-6. DOI: 10.1007/3-540-36387-4 (cited on page 52).
- [67] R. Grosu and S. A. Smolka. “Monte Carlo model checking”. In: *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*. Proceedings. (Apr. 4–8, 2005). Ed. by N. Halbwachs and L. D. Zuck. Vol. 3440. LNCS. Edinburgh, UK: Springer, 2005, pp. 271–286. ISBN: 3-540-25333-5. DOI: 10.1007/978-3-540-31980-1\_18 (cited on page 65).
- [68] L. Grunske. “Specification patterns for probabilistic quality properties”. In: *30th International Conference on Software Engineering, ICSE 2008*. (May 10–18, 2008). Ed. by W. Schäfer, M. B. Dwyer, and V. Gruhn. Leipzig, Germany: ACM, 2008, pp. 31–40. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368094 (cited on page 53).
- [69] D. Guck. *Interactive Markov Chains Analyzer*. English. Version Version 1.2. RWTH Aachen University. 2010. 70 pp. (cited on page 98).
- [70] D. Guck et al. “Quantitative timed analysis of interactive Markov chains”. In: *4th International Symposium on NASA Formal Methods, NFM 2012*. Proceedings. (Apr. 3–5, 2012). Ed. by A. Goodloe and S. Person. Vol. 7226. LNCS. Norfolk, VA, USA: Springer, 2012, pp. 8–23. ISBN: 978-3-642-28890-6. DOI: 10.1007/978-3-642-28891-3\_4 (cited on pages 97, 112, 194).

- [71] A. Guiotto et al. “FAME process: a dedicated development and V&V process for FDIR”. In: *DASIA 2014 – DAta Systems In Aerospace, 3-5 06 2014*. Proceedings. Ed. by L. Ouwehand. Vol. SP-725. Warsaw, Poland: ESA Communications, 2014. ISBN: 978-92-9221-289-6 (cited on page 92).
- [72] Y. Gurevich. “Evolving algebras 1993: Lipari guide”. In: ed. by E. Börger. Oxford University Press, 1995, pp. 9–36. ISBN: 0-19-853854-5 (cited on page 53).
- [73] H. Hansson and B. Jonsson. “A logic for reasoning about time and reliability”. In: *Formal Aspects of Computing* 6.5 (1994), pp. 512–535. DOI: 10.1007/BF01211866 (cited on pages 52, 191).
- [74] D. Harel. “Statecharts: a visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9 (cited on page 126).
- [75] P. Heidelberger. “Fast simulation of rare events in queueing and reliability models”. In: *ACM Transactions on Modeling and Computer Simulation* 5.1 (1995), pp. 43–85. DOI: 10.1145/203091.203094 (cited on page 75).
- [76] D. Henriques et al. “Statistical model checking for Markov decision processes”. In: *9th International Conference on Quantitative Evaluation of Systems, QEST 2012*. (Sept. 17–20, 2012). London, United Kingdom: IEEE Computer Society, 2012, pp. 84–93. ISBN: 978-1-4673-2346-8. DOI: 10.1109/QEST.2012.19 (cited on page 78).
- [77] T. A. Henzinger et al. “Symbolic model checking for real-time systems”. In: *Information and Computation* 111.2 (1994), pp. 193–244. DOI: 10.1006/inco.1994.1045 (cited on page 75).
- [78] T. Héruault et al. “Approximate probabilistic model checking”. In: *5th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2004*. Proceedings. (Jan. 11–13, 2004). Ed. by B. Steffen and G. Levi. Vol. 2937. LNCS. Venice, Italy: Springer, 2004, pp. 73–84. ISBN: 3-540-20803-8. DOI: 10.1007/978-3-540-24622-0\_8 (cited on page 69).
- [79] H. Hermans. *Interactive Markov Chains: The Quest for Quantified Quality*. Vol. 2428. LNCS. Springer, 2002. ISBN: 3-540-44261-8. DOI: 10.1007/3-540-45804-2 (cited on page 40).
- [80] H. Hermans and J.-P. Katoen. “The how and why of interactive Markov chains”. In: *8th International Symposium on Formal Methods for Components and Objects, FMCO 2009*. (Nov. 4–6, 2009). Ed. by F. S. de Boer et al. Vol. 6286. LNCS. Revised Selected Papers. Eindhoven, The Netherlands: Springer, 2010, pp. 311–337. ISBN: 978-3-642-17070-6. DOI: 10.1007/978-3-642-17071-3\_16 (cited on page 40).
- [81] A. B. Ivanov et al. “CubETH: nano-satellite mission for orbit and attitude determination using low-cost GNSS receivers”. In: *Proceedings of the 66th International Astronautical Congress, IAC*. 2015, pp. 3955–3966 (cited on page 144).

- [82] D. Jackson. “Alloy: a lightweight object modelling notation”. In: *ACM Transactions on Software Engineering and Methodology* 11.2 (2002), pp. 256–290. DOI: 10.1145/505145.505149 (cited on page 53).
- [83] C. B. Jones. “Tentative steps toward a development method for interfering programs”. In: *ACM Transactions on Programming Languages and Systems* 5.4 (1983), pp. 596–619. DOI: 10.1145/69575.69577 (cited on page 109).
- [84] S. Junges et al. “One net fits all: a unifying semantics of DFTs using GSPNs”. In: *Proceedings of PetriNets*. LNCS. (To appear). 2018 (cited on page 216).
- [85] B. L. Kaminski et al. “Weakest precondition reasoning for expected runtimes of probabilistic programs”. In: *25th European Symposium on Programming Languages and Systems, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016*. Proceedings. (Apr. 2–8, 2016). Ed. by P. Thiemann. Vol. 9632. LNCS. Eindhoven, The Netherlands: Springer, 2016, pp. 364–389. ISBN: 978-3-662-49497-4. DOI: 10.1007/978-3-662-49498-1\_15 (cited on page 64).
- [86] J.-P. Katoen, L. Song, and L. Zhang. “Probably safe or live”. In: *23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS ’14. Joint Meeting*. (July 14–18, 2014). Ed. by T. A. Henzinger and D. Miller. Vienna, Austria: ACM, 2014, 55:1–55:10. ISBN: 978-1-4503-2886-9. DOI: 10.1145/2603088.2603147 (cited on page 53).
- [87] J.-P. Katoen et al. “The ins and outs of the probabilistic model checker MRMC”. In: *Performance Evaluation* 68.2 (2011), pp. 90–104. DOI: 10.1016/j.peva.2010.04.001 (cited on pages 69, 97).
- [88] E. Kindler. “Safety and liveness properties: a survey”. In: *Bulletin of the European Association for Theoretical Computer Science* 53.268-272 (1994), p. 30 (cited on page 53).
- [89] L. Kof. “Natural language processing: mature enough for requirements documents analysis?” In: *10th International Conference on Applications of Natural Language to Information Systems, NLDB 2005*. Proceedings. (June 15–17, 2005). Ed. by A. Montoyo, R. Muñoz, and E. Métais. Vol. 3513. LNCS. Alicante, Spain: Springer, 2005, pp. 91–102. ISBN: 3-540-26031-5. DOI: 10.1007/11428817\_9 (cited on page 56).
- [90] L. Kof. “Requirements analysis: concept extraction and translation of textual specifications to executable models”. In: *14th International Conference on Applications of Natural Language to Information Systems, NLDB 2009*. (June 24–26, 2009). Ed. by H. Horacek et al. Vol. 5723. LNCS. Revised Papers. Saarbrücken, Germany: Springer, 2010, pp. 79–90. ISBN: 978-3-642-12549-2. DOI: 10.1007/978-3-642-12550-8\_7 (cited on page 56).

- [91] G. Konecny. “Small satellites—a tool for earth observation?” In: *20th International congress for photogrammetry and remote sensing, ISPRS XXth congress, 2004*. Vol. 35. International Archives of Photogrammetry. Istanbul, Turkey: Organising Committee of the XXth international congress for photogrammetry and remote sensing, 2004, pp. 580–582 (cited on page 143).
- [92] S. Konrad and B. H. C. Cheng. “Real-time specification patterns”. In: *27th International Conference on Software Engineering, ICSE 2005*. (May 15–21, 2005). Ed. by G.-C. Roman, W. G. Griswold, and B. Nuseibeh. St. Louis, Missouri, USA: ACM, 2005, pp. 372–381. DOI: 10.1145/1062455.1062526 (cited on page 53).
- [93] G. Kotonya and I. Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley Publishing, 1998. ISBN: 978-0471972082 (cited on page 45).
- [94] R. Koymans. “Specifying real-time properties with metric temporal logic”. In: *Real-Time Systems 2.4* (1990), pp. 255–299. DOI: 10.1007/BF01995674 (cited on pages 52, 192).
- [95] M. Z. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: verification of probabilistic real-time systems”. In: *23rd International Conference on Computer Aided Verification, CAV 2011*. Proceedings. (July 14–20, 2011). Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Snowbird, UT, USA: Springer, 2011, pp. 585–591. ISBN: 978-3-642-22109-5. DOI: 10.1007/978-3-642-22110-1\_47 (cited on page 69).
- [96] S. Lauesen and O. Vinter. “Preventing requirement defects: an experiment in process improvement”. In: *Requirements Engineering 6.1* (2001), pp. 37–50. DOI: 10.1007/PL00010355 (cited on page 45).
- [97] P. Manolios and R. J. Treffler. “Safety and liveness in branching time”. In: *16th Annual IEEE Symposium on Logic in Computer Science*. Proceedings. (June 16–19, 2001). Boston, Massachusetts, USA: IEEE Computer Society, 2001, pp. 366–374. ISBN: 0-7695-1281-X. DOI: 10.1109/LICS.2001.932512 (cited on page 53).
- [98] M. A. Marsan et al. “The effect of execution policies on the semantics and analysis of stochastic Petri nets”. In: *IEEE Transactions on Software Engineering* 15.7 (1989), pp. 832–846. DOI: 10.1109/32.29483 (cited on pages 76, 77).
- [99] S. Matsuda et al. “Development of an affordable and dedicated nano-launcher”. In: *AIAA 8th Responsive Space Conference*. 2010 (cited on page 143).
- [100] C. Mattarei et al. “Comparing different functional allocations in automated air traffic control design”. In: *Formal Methods in Computer-Aided Design, FMCAD 2015*. (Sept. 27–30, 2015). Ed. by R. Kaivola and T. Wahl. Austin, Texas, USA: IEEE Computer Society, 2015, pp. 112–119. ISBN: 978-0-9835678-5-1 (cited on page 123).

- [101] A. Mavridou et al. *Architecture-based Design: A Satellite On-Board Software Case Study*. Technical Report. EPFL, 2016. URL: <https://infoscience.epfl.ch/record/221156> (cited on page 158).
- [102] A. Mavridou et al. “Architecture-based design: a satellite on-board software case study”. In: *13th International Conference on Formal Aspects of Component Software, FACS 2016*. (Oct. 19–21, 2016). Ed. by O. Kouchnarenko and R. Khosravi. Vol. 10231. LNCS. Revised Selected Papers. Besançon, France, 2017, pp. 260–279. ISBN: 978-3-319-57665-7. DOI: 10.1007/978-3-319-57666-4\_16 (cited on pages 144, 147, 150, 163, 179, 182).
- [103] A. Mavridou et al. “Configuration logics: modeling architecture styles”. In: *Journal of Logical and Algebraic Methods in Programming* 86.1 (2017), pp. 2–29. DOI: 10.1016/j.jlamp.2016.05.002 (cited on page 144).
- [104] S. Merz. “Decidability and incompleteness results for first-order temporal logics of linear time”. In: *Journal of Applied Non-Classical Logics* 2.2 (1992), pp. 139–156. DOI: 10.1080/11663081.1992.10510779 (cited on page 57).
- [105] J. F. Meyer. “Performability: a retrospective and some pointers to the future”. In: *Performance Evaluation* 14.3-4 (1992), pp. 139–156. DOI: 10.1016/0166-5316(92)90002-X (cited on page 111).
- [106] A. Misra et al. “Diagnosability of dynamical systems”. In: *3rd International Workshop on Principles of Diagnosis, DX’92*. (Oct. 12–14, 1992). Rosario, WA, USA, 1992 (cited on page 216).
- [107] K. S. Namjoshi et al., eds. *5th International Symposium on Automated Technology for Verification and Analysis, ATVA 2007*. Proceedings. (Oct. 22–25, 2007). Vol. 4762. LNCS. Tokyo, Japan: Springer, 2007. ISBN: 978-3-540-75595-1. DOI: 10.1007/978-3-540-75596-8.
- [108] I. Nason, J. Puig-Suari, and R. Twiggs. “Development of a family of picosatellite deployers based on the CubeSat standard”. In: *IEEE Aerospace Conference Proceedings, 2002*. Vol. 457–464. IEEE. IEEE Computer Society, 2002, pp. 1–1. DOI: 10.1109/AERO.2002.1036865 (cited on page 143).
- [109] R. Nelken and N. Francez. “Automatic translation of natural language system specifications”. In: *8th International Conference on Computer Aided Verification, CAV ’96*. Proceedings. (July 31–Aug. 3, 1996). Ed. by R. Alur and T. A. Henzinger. Vol. 1102. LNCS. New Brunswick, NJ, USA: Springer, 1996, pp. 360–371. ISBN: 3-540-61474-5. DOI: 10.1007/3-540-61474-5\_83 (cited on page 56).
- [110] S. U. Nicholas Metropolis. “The Monte Carlo method”. In: *Journal of the American statistical association* 247 (1949), pp. 335–341 (cited on page 65).

- [111] R. de Nicola and F. W. Vaandrager. “Action versus state based logics for transition systems”. In: *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science*. Proceedings. (Apr. 23–27, 1990). Ed. by I. Guessarian. Vol. 469. LNCS. La Roche Posay, France: Springer, 1990, pp. 407–419. ISBN: 3-540-53479-2. DOI: 10.1007/3-540-53479-2\_17 (cited on page 56).
- [112] M. Odenbrett, V. Y. Nguyen, and T. Noll. “Slicing AADL specifications for model checking”. In: *2nd Symposium on NASA Formal Methods, NFM 2010*. Proceedings. (Apr. 13–15, 2010). Ed. by C. A. Muñoz. Vol. NASA/CP-2010-216215. NASA Conference Proceedings. Washington D.C., USA, 2010, pp. 217–221 (cited on page 71).
- [113] OSATE. URL: <http://osate.org/> (cited on page 122).
- [114] J. Ouaknine and J. Worrell. “On the decidability and complexity of metric temporal logic over finite words”. In: *Logical Methods in Computer Science* 3.1 (2007). DOI: 10.2168/LMCS-3(1:8)2007 (cited on page 53).
- [115] M. Pagnamenta. “Rigorous software design for nano and micro satellites using BIP framework”. Master’s Thesis. Space Center EPFL, 2014 (cited on pages 144, 163, 173, 179).
- [116] A. Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science*. (Oct. 31–Nov. 1, 1977). Providence, Rhode Island, USA: IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cited on pages 52, 53, 190).
- [117] K. Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Inc., 2010. ISBN: 978-3642125775 (cited on page 45).
- [118] K. van der Pol and T. Noll. “Security type checking for MILS-AADL specifications”. In: *International Workshop on MILS: Architecture and Assurance for Secure Systems, Amsterdam, The Netherlands*. 2015 (cited on page 93).
- [119] T. Quatmann, S. Junges, and J.-P. Katoen. “Markov automata with multiple objectives”. In: *29th International Conference on Computer Aided Verification, CAV 2017*. Proceedings, Part I. (July 24–28, 2017). Ed. by R. Majumdar and V. Kuncak. Vol. 10426. LNCS. Heidelberg, Germany: Springer, 2017, pp. 140–159. ISBN: 978-3-319-63386-2. DOI: 10.1007/978-3-319-63387-9\_7 (cited on page 123).
- [120] D. Reijsbergen et al. “On hypothesis testing for statistical model checking”. In: *International Journal on Software Tools for Technology Transfer* 17.4 (Aug. 2015), pp. 377–395. ISSN: 1433-2787. DOI: 10.1007/s10009-014-0350-1 (cited on pages 66, 69).
- [121] G. Rubino and B. Tuffin. *Rare event simulation using Monte Carlo methods*. John Wiley & Sons, 2009 (cited on page 75).

- [122] M. Saito and M. Matsumoto. “SIMD-Oriented fast Mersenne twister: a 128-bit pseudorandom number generator”. In: *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Ed. by A. Keller, S. Heinrich, and H. Niederreiter. Springer, 2008, pp. 607–622. ISBN: 978-3-540-74496-2 (cited on page 82).
- [123] *SAVOIR: Space AVionics Open Interface aRchitecture*. URL: <http://savoir.estec.esa.int/> (cited on page 3).
- [124] S. Sebastio and A. Vandin. “MultiVeStA: statistical model checking for discrete event simulators”. In: *7th International Conference on Performance Evaluation Methodologies and Tools, ValueTools '13*. (Dec. 10–12, 2013). Ed. by A. Horváth et al. Torino, Italy: ICST/ACM, 2013, pp. 310–315. ISBN: 978-1-936968-48-0. DOI: 10.4108/icst.valuetools.2013.254377 (cited on page 69).
- [125] F. Shull et al. “What we have learned about fighting defects”. In: *8th IEEE International Software Metrics Symposium (METRICS 2002)*. (July 4–7, 2002). Ottawa, Canada: IEEE Computer Society, 2002, p. 249. ISBN: 0-7695-1339-5. DOI: 10.1109/METRIC.2002.1011343 (cited on page 45).
- [126] *MathWorks Simulink*. URL: <https://www.mathworks.com/products/simulink.html> (cited on pages 99, 122).
- [127] *Stateflow - MATLAB & Simulink*. URL: <https://www.mathworks.com/products/stateflow.html> (cited on page 126).
- [128] *Choose a Solver - MATLAB & Simulink*. URL: <https://www.mathworks.com/help/simulink/ug/types-of-solvers.html> (cited on page 127).
- [129] J. M. Spivey. *Z Notation - a reference manual (2. ed.)* Prentice Hall International Series in Computer Science. Prentice Hall, 1992. ISBN: 978-0-13-978529-0 (cited on page 53).
- [130] M. Stamatelatos et al. *Fault tree handbook with aerospace applications*. NASA Washington, DC, 2002 (cited on page 214).
- [131] S. von Styp, H. C. Bohnenkamp, and J. Schmaltz. “A conformance testing relation for symbolic timed automata”. In: *8th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS 2010*. Proceedings. (Sept. 8–10, 2010). Ed. by K. Chatterjee and T. A. Henzinger. Vol. 6246. LNCS. Klosterneuburg, Austria: Springer, 2010, pp. 243–255. ISBN: 978-3-642-15296-2. DOI: 10.1007/978-3-642-15297-9\_19 (cited on page 123).
- [132] *SysML*. URL: <http://sysml.org/> (cited on page 122).
- [133] *TASTE: The ASSERT Set of Tools for Engineering*. URL: <https://essr.esa.int/project/taste> (cited on pages 3, 124).
- [134] S. Tripakis. “Verifying progress in timed systems”. In: *5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems, ARTS'99*. Proceedings. (May 26–28, 1999). Ed. by J.-P. Katoen. Vol. 1601. LNCS. Bamberg, Germany: Springer, 1999, pp. 299–314. ISBN: 3-540-66010-0. DOI: 10.1007/3-540-48778-6\_18 (cited on page 74).

- [135] S. Uman and J. von Neumann. “On combination of stochastic and deterministic processes”. In: *Bulletin of the American Mathematical Society* 53 (1947), p. 1120 (cited on page 65).
- [136] W. E. Vesely et al. *Fault Tree Handbook*. Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Commission, Jan. 1981. ISBN: 978-0160055829 (cited on page 214).
- [137] M. Villén-Altamirano and J. Villén-Altamirano. “Restart: a straightforward method for fast simulation of rare events”. In: *26th Conference on Winter simulation, WSC 1994*. Proceedings. (Dec. 11–14, 1994). Ed. by D. A. Sadowski et al. Lake Buena Vista, FL, USA: ACM, 1994, pp. 282–289. ISBN: 0-7803-2109-X. DOI: 10.1109/WSC.1994.717150 (cited on page 76).
- [138] *Voyager - Mission Status*. URL: <https://voyager.jpl.nasa.gov/mission/status/> (cited on page 1).
- [139] *Computers in Spaceflight: The NASA Experience*. URL: <https://history.nasa.gov/computers/Ch6-2.html> (cited on page 1).
- [140] A. Wald. “Sequential tests of statistical hypotheses”. In: *The Annals of Mathematical Statistics* 16.2 (1945), pp. 117–186 (cited on page 67).
- [141] R. Wimmer et al. “Sigref - a symbolic bisimulation tool box”. In: *4th International Symposium on Automated Technology for Verification and Analysis, ATVA 2006*. Proceedings. (Oct. 23–26, 2006). Ed. by S. Graf and W. Zhang. Vol. 4218. LNCS. Beijing, China: Springer, 2006, pp. 477–492. ISBN: 3-540-47237-1. DOI: 10.1007/11901914\_35 (cited on pages 97, 112).
- [142] H. L. S. Younes. “Verification and planning for stochastic processes with asynchronous events”. Ph.D. Thesis. Carnegie-Mellon University Pittsburg Pennsylvania, School of computer science, 2005 (cited on pages 65, 67).
- [143] H. L. S. Younes. “Ymer: a statistical model checker”. In: *17th International Conference on Computer Aided Verification, CAV 2005*. Proceedings. (July 6–10, 2005). Ed. by K. Etessami and S. K. Rajamani. Vol. 3576. LNCS. Edinburgh, Scotland, UK: Springer, pp. 429–433. ISBN: 3-540-27231-3. DOI: 10.1007/11513988\_43 (cited on pages 69, 75).
- [144] I. S. Zapreev and C. Jansen. *Markov Reward Model Checker*. English. Version Version 1.5. RWTH Aachen University. 2011. 70 pp. (cited on page 98).

# STANDARDS

- [EHB1002A] *Verification guidelines*. ECSS Standard ECSS-E-HB-10-02A. European Cooperation for Space Standardization, Dec. 2010 (cited on page 46).
- [EST1006C] *Technical requirements specification*. ECSS Standard ECSS-E-ST-10-06C. European Cooperation for Space Standardization, Dec. 2010 (cited on pages 47, 49, 50).
- [MST10C] *Project planning and implementation*. ECSS Standard ECSS-M-ST-10C Rev.1. European Cooperation for Space Standardization, Mar. 2009 (cited on page 46).
- [OPC] *Information technology – Document description and processing languages – Office Open XML File Formats – Part 2: Open Packaging Conventions*. ISO/IEC 29500-2:2012. Sept. 2012. URL: <https://www.iso.org/standard/61796.html> (cited on page 141).
- [OCL] *Object Constraint Language*. OMG Specification. Object Management Group, Feb. 2014. URL: <https://www.omg.org/spec/OCL/> (cited on page 53).
- [AS5506-1] *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface, Annex E: Error Model Annex*. SAE Standard SAE AS5506-1. International Society of Automotive Engineers, Apr. 2011 (cited on pages 80, 98).
- [AS5506-1A] *Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: ARINC653 Annex, Annex C: Code Generation Annex, Annex E: Error Model Annex*. SAE Standard SAE AS5506-1A. International Society of Automotive Engineers, Sept. 2015 (cited on page 41).
- [AS5506-2] *Architecture Analysis and Design Language (AADL) Annex Volume 2: Annex B: Data Modeling Annex; Annex D: Behavior Model Annex; Annex F: ARINC653 Annex*. SAE Standard SAE AS5506-2. International Society of Automotive Engineers, Jan. 2011 (cited on page 42).
- [AS5506C] *Architectural Analysis & Design Language (AADL) V2.2*. SAE Standard SAE AS5506C. International Society of Automotive Engineers, Jan. 2017 (cited on pages 10, 43).

[SLIMv3] *SLIM 3.0 - Syntax and Semantics*. Manual. COMPASS Consortium, Dec. 2016 (cited on pages 10, 192).

## PRIOR PUBLICATIONS

- [HB1] D. Bohlender et al. “A review of statistical model checking pitfalls on real-time stochastic models”. In: *6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, ISO LA 2014*. Proceedings, Part II. (Oct. 8–11, 2014). Ed. by T. Margaria and B. Steffen. Vol. 8803. LNCS. Imperial, Corfu, Greece: Springer, 2014, pp. 177–192. ISBN: 978-3-662-45230-1. DOI: 10.1007/978-3-662-45231-8\_13 (cited on pages 5, 69, 83).
- [HB2] V. Bos, H. Bruintjes, and S. Tonetta. “Catalogue of system and software properties”. In: *35th International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2016*. Proceedings. (Sept. 21–23, 2016). Ed. by A. Skavhaug, J. Guiochet, and F. Bitsch. Vol. 9922. LNCS. Trondheim, Norway: Springer, 2016, pp. 88–101. ISBN: 978-3-319-45476-4. DOI: 10.1007/978-3-319-45477-1\_8 (cited on pages 5, 59).
- [HB3] M. Bozzano et al. “The COMPASS 3.0 toolset (short paper)”. In: *Proceedings of the 5th International Symposium on Model-Based Safety and Assessment, IMBSA 2017*. 2017. URL: <https://drive.google.com/open?id=oB9DzO9PFER2xRElXeGh6X1pjS1k> (cited on page 6).
- [HB4] H. Bruintjes, J.-P. Katoen, and D. Lesens. “A statistical approach for timed reachability in AADL models”. In: *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015*. (June 22–25, 2015). Rio de Janeiro, Brazil: IEEE Computer Society, 2015, pp. 81–88. ISBN: 978-1-4799-8629-3. DOI: 10.1109/DSN.2015.32 (cited on pages 6, 79).
- [HB5] C. Dehnert et al. “PROPhESY: a probabilistic parameter synthesis tool”. In: *Computer Aided Verification - 27th International Conference, CAV 2015*. Proceedings, Part I. (July 18–24, 2015). Ed. by D. Kroening and C. S. Pasareanu. Vol. 9206. LNCS. San Francisco, CA, USA: Springer, 2015, pp. 214–231. ISBN: 978-3-319-21689-8. DOI: 10.1007/978-3-319-21690-4\_13 (cited on page 123).
- [HB6] C. Dehnert et al. “Parameter synthesis for probabilistic systems”. In: *19th GI/ITG/GMM Workshop on Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016*. (Mar. 1–2, 2016). Ed. by R. Wimmer. Freiburg im Breisgau, Germany: Albert-Ludwigs-Universität Freiburg, 2016, pp. 72–74. ISBN: 978-3-00-052380-9. DOI: 10.6094/UNIFR/10639.

