

ULRR

Synthesis of software design models

Item Type	Thesis
Authors	Shokry, Hesham
Download date	2026-06-06 16:42:04
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/3612

Synthesis of Software Design Models



UNIVERSITY of LIMERICK
OLLSCOIL LUIMNIGH

University of Limerick

Thesis submitted to the Department of Computer Science and Information Systems, Faculty of

Science and Engineering, University of Limerick

In fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

December 2013

Hesham Shokry Ali Abdallah, M.Sc.

Supervisor: Prof. Mike Hinchey

Synthesis of Software Design Models

Approved by
Dissertation Committee:

DEDICATION

To the thirty thousands brave men and women, who defended Tahrir square and the Egyptian Revolution in the night of the 1st February 2011.

To the faithful thousands that have been killed in Rabaa square defending their choice and freedom on the 14th August 2013.

To those who defend freedom anywhere.

Acknowledgements

First, my sincere thanks to Professor Mike Hinchey, my PhD advisor, for providing me the opportunity for carrying out the research leading to this thesis. Mike's encouragement and advice during the selection of the topic, as well as during the work on it that followed, were very valuable. Mike's professional supervision style and his continuous advice has paved the way to achieve it. Thank you Mike!

My thanks also go to Dr. Ita Richardson and Dr. Goetz Botterweck for their occasional advices that helped me to frame questions and apply proper research methodology and evaluations in this thesis. The wisdom of Dr. Norah Power have always been a guiding light to my work and kept me aware of the big picture. Thank you very much Ita, Goetz and Norah.

I am especially indebted to Prof. David Parnas who voluntarily responded to my inquires about his research work of "modes". Dave's extremely encouraging and scientifically provocative responses helped me to critically think of the compelling engineering research problems to solve. Thank you Dave for this invaluable input.

The quality and presentation of this thesis would not have come to this level without the contributions from Dr. Jim Buckley who consistently (and very patiently) reviewed the thesis content in detail in several rounds. Jim's comprehensive feedback helpedme reorganize the presentation flow and structure. Thank you very much, Jim.

There are many other people who have indirectly contributed to this work: Patsy, Edel, Suzan and Dara from Lero's admin office have facilitated all the paper work and communications with UL. Denis and Ger from Lero's IT team have been always responding to my (too many) IT requests and rescued me on many occasions! Also, my PhD journey would not be so exciting without the company of my PhD colleagues Sarmad, Lianping, Anila, Pdraig, Klass, Saad and Jacek. Thanks you all.

Finally, I am so grateful to my wife, Rehab Ramadan, who stood firmly beside me on every stage of my PhD journey. Thank you Rehab for making it easier for me, particularly in the early stages when I was wandering in all directions. Thanks for sacrificing many holidays, looking after the little

ones Nayera and Faris, to give me the time needed.

Hesham Shokry Ali Abdallah

University of Limerick

December 2013

Abstract

Early system requirements are often captured by declarative and property-based artefacts, such as scenarios and goals. While such artifacts are intuitive and useful, they are partial and typically lack an overarching structure to allow systematic elaboration of the partial behaviors they denote. We propose a structuring approach appropriate for scoping different partial behaviors, focusing on scenario-based behavior specifications. The approach is based on Parnas' notions of 'modes' and 'mode-classes', where a mode is a set of states that satisfy some predicate, and a mode-class is a collection of disjoint modes that partitions the system's state-space so that each state belongs to exactly one mode. There may be several mode-classes, in which case every state belongs to exactly one mode from each mode-class. We structure a scenario by partitioning its observed states into modes, allowing elaboration of the scenario's parts independently without losing the overall system view. Having every scenario partitioned via a suitable mode-class, we merge the mode-classes constructively to build a single behavioural model of the system. The evidence presented here suggests that this facilitates early refinement and an improved coverage of requirements, as well as improved generation of system models from partial behaviors. We provide a sound formal model of modes, based on which we detail a novel technique to synthesize a prototype of system behavior, given a set of scenarios and corresponding mode-classes specifications as input.

Contents

Acknowledgements	iv
Abstract	v
List of Tables	xi
List of Figures	xii
Chapter 1 Introduction and Research Strategy	1
1.1 Introduction	1
1.1.1 Prototyping vs Synthesis	1
1.1.2 Synthesis in Software Engineering	2
1.2 Research Challenges	5
1.3 Research Plan	6
1.3.1 Research Hypotheses	6
1.3.2 Research Questions	7
1.3.3 Research Strategy and Evaluation Plan	8
1.4 Structure of this Thesis	9
Chapter 2 Interactions-Based Requirements Specifications	10
2.1 Introduction	11
2.1.1 Basics and Terminology	11
2.1.2 Prototyping vs. Synthesis	12
2.2 Running Example	14
2.3 Interactions-Based Specifications	14
2.3.1 MSC-96	16

2.3.2	LSC	29
2.3.3	OMSC	30
2.3.4	HySC	32
2.4	Other Forms of Interaction-Based Specifications	32
2.5	Chapter Summary	33
Chapter 3 Software Behavior Model Synthesis		34
3.1	Introduction	34
3.2	Background	36
3.2.1	Models <i>and</i> Software Behavior Models	36
3.2.2	Prototyping and Synthesis	38
3.3	Behavior Models Synthesis – State-of-the-Art	40
3.3.1	Input to the Synthesis Process	40
3.3.2	Generated Models	44
3.3.3	The Methodical Aspects of Synthesis Process	47
3.3.4	State Space coverage	50
3.4	Discussion and Summary	50
Chapter 4 Behavior Modality		52
4.1	The origins of modes in Hybrid Systems	52
4.1.1	Behavior Hybridism and the Role of Modes in Hybrid Systems	53
4.1.2	Moving From Hybrid Physical Processes to Technological Systems	55
4.1.3	N-Degrees of Modality in Behavior: Engineering Software with Modes	56
4.1.4	Section Summary	58
4.2	Achieving N-Degrees of Modality via N-Classes of Modes	58
4.2.1	What makes a good abstraction	59
4.2.2	Mode-Classes: Reinventing “Modes” in Software Engineering	59
4.3	Automata-Theoretic Formulation of Modes and Mode-Classes	60
4.3.1	Mode vs State	60
4.3.2	Mode-Class	61
4.3.3	Mode-Machine	63
4.3.4	Mode Machine Refinement	65
4.4	Modes Identification and Selection	66
4.4.1	Canonical Form of Mode-Classes	67

4.4.2	Modal vs Operational Behavior	75
4.5	Discussion and Summary	76
Chapter 5 Structured Scenario Specifications		77
5.1	Introduction	78
5.2	Scenarios Structuring: State-of-the-Art	80
5.2.1	High Level Message Sequence Charts HMSC	81
5.2.2	UML Interaction Overview Diagrams IOD	85
5.2.3	Early Aspects	88
5.2.4	Other Approaches to Realizing Inter-Scenario Relationships	89
5.2.5	Concluding Summary	90
5.3	Mode-Wise Structuring of Scenarios	91
5.3.1	The Synergy Between Modes and Scenarios	92
5.3.2	Scenarios Identification and Structuring	94
5.4	Case Studies	96
5.5	First case study: Steam Boiler Controller	97
5.5.1	The Original Requirements of SBC	97
5.5.2	Naive Scenario Specifications of SBC	101
5.5.3	Structuring the SBC Scenarios	106
5.6	Second case study: Mine Pump Controller MPC	115
5.6.1	The Original Specifications of MPC	115
5.6.2	Structuring the MPC Scenarios	117
5.7	Chapter Summary	129
Chapter 6 Synthesizing Automata Models From Structured Scenarios		131
6.1	New Process for Behavior Model Synthesis	132
6.1.1	Process Phases	133
6.1.2	Observations and Discussion	139
6.2	Case Studies	140
6.2.1	ESFAS	140
6.3	Related Work	143
6.4	Conclusions	145

Chapter 7 Summary and Outlook	147
7.1 Summary	147
7.1.1 <i>Synthesis-friendly specifications:</i>	147
7.1.2 <i>Modes:</i>	148
7.1.3 <i>State variables:</i>	150
7.1.4 <i>State-based versus Event-based automata:</i>	150
7.1.5 <i>Distinguished types of Behaviors:</i>	151
7.1.6 <i>Dealing with Complexity:</i>	151
7.1.7 <i>Synthesizing analyzable models:</i>	151
7.1.8 Research validation and results	152
7.2 Research Findings	153
7.2.1 Hypothesis 1: The provision of distinct contexts of the system allows us to scope and structure partial specifications, such as scenarios, and enables better coverage of the requirements	153
7.2.2 Hypothesis 2: Structuring the partial requirements specifications would result in a more adequate input to synthesis techniques so as to avoid known issues in existing synthesis techniques.	155
7.3 Outlook	157
Bibliography	157

List of Tables

4.1	State variables of the ESFAS controller	72
5.1	Messages sent by the SBC	102
5.2	Messages received by the SBC	103
5.3	The identified state-variables of the Steam Boiler Controller	108
5.4	Predicate functions of the transitions in the mode machine MM_{fm}	110
5.5	Predicate functions of the transitions in the mode machine MM_{fm}	112
5.6	Messages received by the MPC	118
5.7	Messages sent by the MPC	118
5.8	The identified state-variables of the Mine Pump Controller	121

List of Figures

2.1	Example of Basic MSC diagram	17
2.2	Example of the environment frame in MSC	20
2.3	Example of Conditions and Actions in MSC	21
2.4	Example of coregions and general orderings in MSC	25
4.1	Hybrid System example: bouncing ball behavior trajectory	54
4.2	Technological System example: Temperature Controller	55
4.3	Behavior Irregularity Spectrum	57
4.4	Pictorial illustration of state-space partitioning.	62
4.5	(a) Illustration of the ESFAS mode-machines MM^{pr} and MM^{sb} . (b) Illustration of the MM^{pr} mode-machine (in-dotted-lines) refined with the FSMs (converted from the scoped scenarios).	64
4.6	Bouncing ball mode-classes	71
4.7	Bouncing ball's position mode-class $MC_{BallMotion} = \langle M_{up}^{BallMotion}, M_{down}^{BallMotion} \rangle$. .	71
4.8	Canonical Mode-Classes of the ESFAS Controller: (a) MC_{sb} Mode-Class of the push-button sb , (b) MC_{ss} Mode-Class of the safety-signal status variable ss , and (c) MC_{pr} Mode-Class of the pressure variable pr	73
4.9	ESFAS mode-classes after merging and refinement to a state-level model.	74
5.1	Examples of composition structures in HMSC: sequential, unbounded repetition and alternative	83
5.2	(a) Example of parallel composition HMSC. (b) illustration of Connections Nodes in HMSC	84
5.3	UML Interactive Overview Diagram features: Activity Diagram (left) and basic Sequence Diagram SD (right)	87

5.4	Schematic diagram of the Steam Boiler system	98
5.5	Context diagram of the Steam Boiler system	99
5.6	SBC initialization scenario (unstructured)	104
5.7	SBC behavior in Normal mode	105
5.8	Failure Modes Machine MM_{fm}	110
5.9	<i>Init-1</i> and <i>Norm-1</i> scenarios specified under the scope of the modes <i>Initialization</i> and <i>Normal</i> , respectively. These scenarios are shorter than their unstructured version in Figures 5.6 and 5.7 specified directly from initial specifications.	111
5.10	Schematic Diagram of the Mine Pump	116
5.11	Naive scenario of MPC behavior (adapted from [133])	119
5.12	Context diagram of the Mine Pump system	120
5.13	The MPC's Modal Behavior Behavior-3 (<i>pump control on appearance and absence of Methane-level condition</i>).	123
5.14	scenarios scoped by modes in the mode class MC_{meth} : (a) normal water-level scenario under scope of M_1^{meth} , (b) Alarm activation scenario under scope of M_1^{meth} too, (c) water-level control under scope of M_2^{meth}	124
5.15	The MPC's Modal Behavior Behavior-4 of <i>pump failure condition</i>	125
5.16	Scenarios scoped by MC_{ps} : (a) normal water-level control scenario in M_1^{ps} , (b) water-level control in M_2^{ps}	126
5.17	The MPC's Modal Behavior of <i>change of control authority</i>	127
5.18	Water level control scenarios: (a) scenarios under scope of M_1^{ca} , (b) scenarios under scope of M_2^{ca}	128
6.1	Behavior Synthesis Process (using micro-sized versions of ESFAS example.)	134
6.2	Illustration of the synthesis procedure (a) modes M_q^i and M_r^j overlapping (b) M_q^i and M_r^j after intersection (c) transition elaboration.	137
6.3	mode-machines in (a) MM^{pr} and (b) MM^{sb} are shown in dotted lines and their transition relations are elaborated to link the FSMs (solid lines). The red-color indicates the discovered prohibited transitions.	142

Chapter 1

Introduction and Research Strategy

1.1 Introduction

Software requirements development is a key engineering activity that takes place at early stages of software engineering projects. Design and implementations are typically stage-gated by the output of requirements development. Software engineers discover requirements inconsistencies too late in the development cycle, and modifying code at this stage significantly affects the project schedule and budget. To avoid these situations, engineers adopt semi-automated analysis to show inconsistencies in early requirements specification.

Different strategies exist, such as adopting incremental development techniques or building an early prototype of the software system for checking the possible mismatches and inconsistencies in requirements and also for design exploration purposes. This thesis is concerned with CASE¹ techniques to help software engineers to discover inconsistencies in the given specifications of the system at early stages of the development lifecycle. These techniques are typically semi-automated and are typically used to generate analyzable output models of the system under development.

1.1.1 Prototyping vs Synthesis

A wide range of CASE techniques have been proposed to help in modeling software systems. Popular examples include informal techniques such as UML [149] and MSC [72] as well as more formal and theoretically founded techniques such as State Machines and Tabular Specification [7, 63] methods. In software engineering, these techniques are applied in two major areas, namely *Prototyping*

¹Computer Aided Software Engineering

and *Synthesis*, and they have been actively described in the computing literature.

A system *prototype* is a mock-up of the system and is typically built using technology that is different from the one used in final realization of the system (e.g., a different programming language). The prototype mimics general properties of the system-to-be, such as the user interface, look-and-feel, etc. and is used for user evaluations and feedback.

Synthesis, on the other hand, refers to generating a more elaborate model of the system, closer to the final realization of the system than a prototype. *Synthesis* is commonly understood in digital hardware design as the process of taking a high-level description of the system as input and turning it into a lower-level description; for example a silicon layout is synthesized from an algorithmic description (e.g., Verilog) of the Integrated Circuits chips. In general, Synthesis in digital hardware development is meant to be systematic and to have little or no manual interference.

1.1.2 Synthesis in Software Engineering

Synthesis is commonly understood in software engineering as a technical process to automate the generating of an implementation artefact from a given design model. The concept of *synthesis* in software engineering originally emerged in the development of compilers for computer languages. In compiler terms, executable code generation is well known as the *machine-code synthesis* phase of the compilation process [5]. This usage is consistent with the case of digital hardware design, where the output of the synthesis algorithm is an end implementation (or realization).

Inspired by (automated) machine-code generation in compilers, some CASE tools have emerged recently in industry to automate the generation of source-code from specific models [128]. Even though the source-code artefact is not the final implementation of the system, it has contributed to the automation of the development cycle and it is assumed to tie in directly into compilers to build a final implementation. Successful to some extent, this attempt has brought the power of *synthesis* techniques up one level in the development cycle, compared to machine-code generation in compilers. TargetLink® (www.dspaceInc.com) is an example of these tools where C language source-code is generated from synchronous data flow models such as Simulink® models (www.mathworks.com). Not surprisingly; these *source-code synthesis* tools stand on top of a lot of work on code *auto-generation engines* that have made their way into safety-critical applications [126]. Other example tools include IBM Rational tools (www.ibm.com) and the SCADE tool (www.esterel-technologies.com).

This trend of automating software engineering has recently made its way to earlier phases in the development cycle. Recent research efforts have focused on automating the design task by synthesizing design models of the system. Generally speaking, we can think of a *design model* as a

not-fully-thought-through implementation of the system. Those not-fully-thought-through parts in the model represent the high-level decisions about the observable behavior of the system, e.g., how the system responds to external stimuli so that it satisfies (or does not violate) the system requirements. The other “half” represent the implementation decisions. Automating the task of building such models is the focus of this thesis.

In general, the task of synthesizing design-models accepts a set of requirements specifications as input, and outputs an integrated behavioral model that have the following attributes: (1) exhibits (at least) all the behaviors implied in the input requirements and (2) satisfies all the constraints assumed by these requirements. More behaviors may well be introduced in the output model during the synthesis process. Such additional information is a result of elaboration of requirements discovered and/or decisions made by engineers to satisfy some design constraints.

The resulting models, besides serving as early prototypes for validating the requirements, are also used for a variety of other purposes. For example, formal analysis techniques such as Model Checking [33] can be applied to synthesized models for a comprehensive check of system-level properties such as safety and liveness. Moreover, the synthesized models can be used as test oracles in the verification activities during the development cycle. A more useful application of the synthesized model is to auto-generate source-code directly from them, building on off-the-shelf code-generation.

Looking at the state-of-practice, software requirements specifications are often captured via declarative specifications, such as Properties [77] and Goals [11, 90], and scenario-based specifications [72, 103]. These forms of specifications are inherently partial statements that describe assertions or sequences of actions that are incomplete with respect to the behavior of the system-to-be. Consequently, this incompleteness results in inadequate coverage of the system behavior space.

That is a major limitation of synthesis approaches is that the models being synthesized are assumed to be complete descriptions of the system behavior with respect to the input specifications. Given the partial nature of the synthesis inputs (e.g., properties and scenarios), it is sometimes difficult to synthesize an integrated behavioral model (e.g., the problem to find a Least Refinement Model in [133]).

Existing work has attempted a semi-automatic synthesis of system-level (e.g., [132]) and component-level models (e.g., [83, 146]) from partial specifications. A common characteristic of these approaches is that they address the symptom of the partiality problem instead of its origin which is the lack of a proper overarching structure for the specifications. In other words, they accept unstructured and partial scenario-based specifications without attempting to organize them and reduce their partiality, before feeding them to the synthesis process.

A lack of such contextual information leaves system designers with no option other than ad hoc specifications, and no criteria to decide when (i.e., at which stage) to start a scenario and when to stop it. This leads to implicit traversing of inter-contexts, which in turn impedes the opportunities for elaborating the individual behaviors and to discover more requirements.

This thesis addresses the partiality issues in requirements specifications and proposes a specification framework based on the notion of modes and mode-classes [7, 63] and a novel technique to synthesize state-based models from structured scenario-based specifications.

We focus on the class of systems known as *reactive systems*. A reactive system, as opposed to a transformational system, is one that keeps an ongoing input/output relationship with its environment. While the purpose of a transformational system is essentially to transform input data into output, reactive systems are intended to interact continuously with their environment and exist in different states at different points-in-time. In response to environmental stimuli, a reactive system adapts its state, performs some action, and gets ready to react to the next stimulus. Such systems may respond differently to the first or the second occurrence of the same stimulus. Transformational systems, on the other, hand are stateless; they always deliver the same data when fed with the same input. We focus on reactive systems because: 1) they are pervasive in our society, 2) their number and complexity is increasing rapidly, and 3) humans are more and more dependent on them and their quality.

Take the example of automotive systems. The automotive industry is embedding ever more software into modern cars. Automotive software includes several safety-critical, real-time systems such as parking-assistance systems, cruise-control systems, engine-control and so forth. In this industry sector, distribution of processing is a matter of fact conceptually and physically. Fixing design flaws in such (reactive) distributed systems is often very expensive. Additionally, some systems might not tolerate bugs at all if human lives are at stake. Another emerging technology that affects our society is Smart Cities [81] where major international corporates, such as IBM, have been embarking on promising initiatives for researching new frontiers to enable embedded intelligence in metropolitan and suburban areas. The supporting infrastructure involves a larger number of embedded computing devices that exchange information and events among each other, as well as with their surrounding environment, in various numbers of conditions. Modeling and understanding the behaviors of these devices, either for each single system or for all of them as a distributed system, is a challenge due to the high reactivity required and the huge number of possible conditions. In order to comprehend the required behavior of such systems, engineers often seek some abstraction of this behavior. Users and system experts often find difficulty in communicating the required behavior, and more challenging difficulties to provide a complete specification of this behavior.

To this end, our work in this thesis attempts to address challenging engineering problems at early stage of software systems development. First we address the partiality problem in requirements specifications by introducing a novel method to structure requirements specified as scenarios. Second, we propose a semi-automated approach for synthesizing a design model from the structured specifications. This chapter pinpoints the research questions of our work and the associated research methodology.

1.2 Research Challenges

Increasing demands on cost-effective development of software systems has given rise to more challenges in software engineering. According to recent studies [98], a key reason behind failures in software projects is due to the late discovery of issues in requirements. One promising direction to avoid this problem is to auto-generate, through a synthesis process, an early prototype of the system so that analysis techniques can be applied to the model to detect possible issues in requirements specifications, which would be very expensive if detected at late stages of development.

There are two major challenges in design model synthesis. First, the synthesized model should serve as a system prototype and also as a design artefact that is extensible to serve further as input to the implementation phase. This requires a number of quality characteristics of the model and poses the challenge of achieving them via the synthesis procedure.

The characteristics of the output models are important for the subsequent analysis and uses of synthesized models. The output model must be: (1) executable: to be amenable for automated analysis techniques (e.g., Model Checking), (2) abstract: to be as high-level as suitable for inspection by engineers and also to hide implementation-specific details at this stage of development, (3) refinable: so that it can be incrementally detailed (possibly automatically) towards a specific platform, allowing to exploit auto-coding technologies for improving the productivity and maintenance of final implementation; and (4) fidelity to requirements: which means that the model should exhibit all behaviors specified in the input specifications and it does not over-describe the behaviors by implying other behaviors (c.f., [135]) unless the system designer explicitly adds those behaviors.

Second: Partiality and fragmentation are barriers to the synthesis process for systematically generating an integrated behavioral model (e.g., state-based model) of the system. Partiality has been one of the barriers in the synthesis process. For instance, some approaches restrict the scenarios to start at the initial state [133], which is not necessarily the case for all possible scenarios. This is a restrictive assumption on the specifications because designers would need more flexibility to capture

system requirements starting at any possible state. Other difficulties in synthesis have arisen when merging the independently-generated behaviors (i.e., automata models) from fragmented specifications. For example, the unavailability of a common refinement [132, 133] that integrates together the behaviors implied in the input specifications. This is mainly due to mismatches between the different pieces of specifications, which can not be detected intuitively from the initial fragmented form of requirements.

Note that composing partial specifications (e.g., by synthesis) could lead to ambiguity as well as cases where undesirable and/or “do not care” behavior arise. The latter issues have been already addressed in the literature (c.f., implied behavior [135]) and we do not address them here.

1.3 Research Plan

Software engineering research seeks better ways to develop and evaluate software systems. Typically, research is motivated by practical problems and targeting key objectives such as quality cost and timeliness of software products. In this section, first we discuss the research questions addressing the challenges discussed in Section 1.1.2, and then we describe our strategy to research and evaluate appropriate solutions advancing the state-of-the-art work done in this area.

1.3.1 Research Hypotheses

According to the categorization of the types of software engineering research questions [124, 125], our research questions are lying in the “Methods and Means of Development” and “Generalization and Characterization” categories [124]. In this section, we frame the research question we envision to address the challenges identified in Section 1.2. Then we describe our strategy to research possible solutions along with an evaluation plan.

To address the challenges mentioned in the previous section, we set out the following hypotheses to research in this thesis:

1. **Hypothesis 1: The provision of distinct contexts of the system allows us to scope and structure partial specifications, such as scenarios, and enables better coverage of requirements.**
2. **Hypothesis 2: Structuring the partial requirements specifications would result in a more adequate input to synthesis techniques so as to avoid known issues in existing synthesis techniques.**

The issues in Hypothesis 2 include difficulties in finding a common integrated behavioral model of the system (c.f., [132, 133]), and also the rigid assumptions made on scenario-based specifications such as the assumption that a scenario must start at the initial state of the system (c.f., [83]).

1.3.2 Research Questions

In order to test and evaluate these hypotheses, we framed the research questions below so as to pinpoint the specific research directions.

The first research question addresses the challenge to ameliorate requirements partiality and fragmentation, in Hypothesis 1. The question seeks to identify the basic features of a structuring framework that could capture the different pieces of requirements and relate them in a cohesive way. In the following discussion, structured specifications will refer to the requirements specifications captured by that framework.

Q-I: What are the characteristics of a specification-framework that is capable of capturing partial requirements in a form suitable as input to the synthesis process?

- **Q-I.a. How can such specification-framework reduce partiality of requirements?**
- **Q-I.b. How can this framework relate the fragmented requirements together so as to facilitate their integration during the synthesis process?**

The second question investigates how the use of structured specifications will improve the design synthesis task and will achieve the necessary qualities of synthesized output models:

Q-II: What improvements can a synthesis process that accepts structured requirements specifications as input achieve?

- **Q-II.a. To what extent can the synthesis process guarantee the existence of a cohesive and integrated output model?**
- **Q-II.b. What assumptions are necessary for the synthesis process (e.g., assumptions in the requirements specifications such as in [133]), and how constraining will these assumptions be?**

These questions are the main drivers for the research work presented in this thesis.

1.3.3 Research Strategy and Evaluation Plan

The most common research strategy solves some aspect of a software development problem by producing a new procedure or technique and validating it through analysis or discussion of a related application of its use [124]. Another common research strategy provides a way to analyze some aspect of software engineering by developing an analytic, often formal, model and validating it through formal analysis or experience with its use. Shaw [124] lists a set of strategies, commonly used in software engineering research, based on the types of the research questions and the nature of results produced by this research. Shaw also identifies the evaluation approaches associated with each type of result.

Based on Shaw's characterization of the software engineering research our research plan was as follows:

First, following the traditional narrative approach of surveying the following segments of literature: (1) Literature related to the software behavior synthesis techniques, and (2) Literature related to requirements specification approaches. We surveyed significant contributions in behavior synthesis techniques and analyzed the experiences reported in different approaches. The reported difficulties provided the major motivation behind the research work in this thesis. We also surveyed and studied existing approaches to behavior modeling, particularly the ones concerned with system-level, as opposed to component-level, modeling. The purpose was to reuse one approach to support structuring of requirements specifications as discussed before.

Second, developing an analytic model for structuring of requirements specifications. We formalized the concept of state-space partitioning that is based on the idea of mode-classes [7, 63] and used this formalization to derive a framework for structuring requirements specifications in the system's state-space. We rely on the rigor of our formalization to provide an evaluation of the proposed framework, which is consistent with the research evaluation strategies reported in [124].

Third, developing a synthesis procedure that generates automata-based models from structured specifications. This synthesis procedure serves two purposes: First, it provides an evaluation of the usefulness of the structuring framework. Second, the procedure itself is a contribution given that its associated algorithms are efficient and intuitive for implementation. We rely on computational complexity techniques as evaluation of the efficiency of our synthesis procedure.

Finally, we use case studies as the main vehicles of communicating our experiences of applying our approach to structuring partial requirements and the subsequent application of our synthesis procedure to generate a behavior model of the system.

1.4 Structure of this Thesis

The rest of this thesis is structured as follows: Chapter 2 provides a detailed overview of the anatomy of scenarios and the various constructs of scenario models. Where necessary, we inline a discussion to elaborate on these constructs and discuss how synthesis-friendly are they. Chapter 3 complements the discussion in Chapter 2 and surveys the state-of-the-art spectrum of synthesis approaches. We identify the problems reported on each of these approaches and build up a plan of how to avoid them in our improved synthesis approach. Chapters 2 and 3 essentially define the problems to which we are going to propose solutions in Chapters 4, 5 and 6.

Starting in Chapter 4 we provide foundational work to elaborate on and define the concepts of *modes* and *mode-machines*. This establishes a substrate for subsequent work where we propose and evaluate a novel methodical development approach. In Chapter 5, we propose the first part of the approach: a technique and procedure for a structured scenario-based requirements specification, accompanied by a case study to illustrate the ideas and concepts involved. In Chapter 6 we introduce the second part of our approach where we further use *modes* to propose a novel automation technique and a synthesis algorithm to generate an integrated state-based model by merging the structured specifications into a single state machine amenable to analysis techniques such as Model Checking. Finally, we conclude in Chapter 7 with a recap of the results and findings in this thesis, and provide an outlook for future work.

Chapter 2

Interactions-Based Requirements Specifications

Prototyping of software behavior has been studied extensively in the literature [31, 99, 100, 131, 138, 147] as a means for capturing the “dynamic” aspect of requirements in some executable (or animated) model. A plethora of approaches have been explored to specify behavior of software systems, and they vary diversely in terms of the target class of the system, level of abstraction, etc. In this thesis we adopt interaction-oriented specification techniques that are popular in research and industry for specifying complex behavior of *reactive systems* [4]. These techniques, in general, specify the system requirements in terms of the exchange of actions and data between the system and its environment. Interaction specifications take several forms, and the scenario-based form is the most popular.

In particular, we are interested in those interactions-based specifications that are, on the one hand (1) suitable for early stages of development where less knowledge about the system is available, such that the specifications type is platform-independent and amenable to further elaboration and elicitation. On the other hand, (2) the specifications type must be *synthesis-friendly* (i.e., suitable for input to automated synthesis techniques) which is the subject matter of this thesis.

In this Chapter, we survey several categories of interaction-oriented specifications, focusing more on scenario-based forms – given that we assume scenario-based specification as input to our synthesis process (Chapter 6). To this extent, we take a standardized and popular dialect of scenarios, Message Sequence Charts MSC [72] and detail down its syntactic features.

2.1 Introduction

Requirements engineering is intended to frame the domain problem, and the resulting specifications are used as input to the subsequent development activities where a software solution is designed and realized in some implementation. Ideally, requirements specifications define a black-box relation between the system's input and output variables. Essentially, the specifications prescribe the required behavior of the system, observed at its external boundaries. The inputs/outputs relation of nontrivial software systems is too complex, and high sophistication is needed to completely and consistently articulate and express these specifications.

A wealth of approaches have been proposed to address the complexity of software specifications, covering a diverse spectrum of sophisticated techniques and notations, and many of these approaches are backed by some theoretical foundation. In this chapter we discuss a subset of those approaches that express the specifications with syntactic and semantic features consumable by automated synthesis techniques. We define the desired characteristics of synthesis-friendly specifications and evaluate the surveyed specification techniques with respect to these characteristics. We start our discussion with a brief overview of *System Prototyping*, a classic concept closely related to early specifications and verification of the system, and then we make a distinction between Prototyping and Synthesis where the later concept is central of this thesis work.

2.1.1 Basics and Terminology

Software development processes, as other engineering disciplines, can be conceptualized as a *Problem/Solution/Realization* model similar to the standard SEI CMMi model¹. We briefly discuss here a projection of this metaphor on to software engineering and informally give general definitions of common terms that we use in this thesis. Although these terms are quite common in the computing literature they are used casually and no there is consensus on the distinction between them. Therefore, the definitions we give here, though informal, shall help to prevent ambiguity and confusion when these terms are used in the same discussion or context.

Problem: is the set of technical and non-technical requirements statements made by system analysts and customers to characterize the problem that the system is intended to solve.

It frames the problem to be solved via some software solution. In theory, the *problem specification space* might have several instances of the problem specifications, identifying different possible *frames*

¹www.sei.cmu.edu/cmmi/

or *perspectives* of the problem. It is those decisions taken by analysts that determine which problem frame is to be engineered in a real system. In this thesis we always associate the term ‘specifications’ to refer to the requirements artefacts, unless mentioned otherwise.

Solution: is the set of decisions made by system designers to characterize a software solution to the specified problem.

Analogous to the Problem Specification tasks, the Solution Description frames out a solution design to be realized by subsequent activities. In the *Problem/Solution/Realization* metaphor, the solution space has several instances of the design description, each of which is typically referred to as *Design Alternatives* [32], and the selection between these alternatives is driven by the upstream requirements constraints. A similar statement can be made on the realization space. In this thesis we always intend the term ‘description’ to refer to the design artefacts, unless otherwise stated.

Realization: is the construction of an implementation of the solution, which is executable on some platform. There may be several possible implementations, and they may vary according to the specifics of the underlying platform.

Over-specification: the requirements artefact is said to be *over-specified* if it is biased on specific solution detail. A similar situation could arise with the design artefact itself when it involves implementation-specific or platform-specific details.

This definition is based on the ideal *Problem/Solution/Realization* metaphor. However, it may be unavoidable to include, for example, solution-related details in the requirements artefact for purposes of optimization, or sometimes the platform details themselves are intentionally included as part of the requirements in a form of constraints of “non-functional” specifications.

Partial Specification: In this case the requirements artefact specifies only a subset of the system behavior. This situation is generally unavoidable in requirements specifications. Typically, system analysts bounce between specifications and analysis tasks to further elaborate and elicit more requirements until they are satisfied with the specifications. This is an ongoing issue with interaction-based specification artefact.

2.1.2 Prototyping vs. Synthesis

The distinction between the prototyping and synthesis concepts lies in the target of the method. Models developed for evaluation purposes, such as discovering inconsistencies and uncertainties in the problems specifications, are commonly referred to as *Prototypes*. These models (or prototypes) are useful for customer demos or as test oracles. It depends on the application domain whether these models are used as a target executable system or they are planned to be just throw-away models.

Methods used for building such models are drawn primarily from Rapid Application Development (RAD) technologies [17, 31, 96] and similar techniques. While most prototypes are executable, their executions are just simulations of the real operation of the target system and the results do help with the solution (i.e., design) decisions – though any implementation must comply with these simulation results.

On the other hand, models developed to address solution-related concerns are intended for purposes beyond those of prototyping. These models are primarily intended for purposes such as evaluating design alternatives and collecting implementation-specific measures such as performance, reliability, etc. To this extent, such models are commonly referred to as Design Models (c.f., [18, 37, 60, 83, 94, 133]). The representations, i.e., model description and formalisms of Design Models, are typically refine-able and amenable to automated implementations through, for example, code generation engines. Whereas the representation of a prototype typically uses programming-like technologies (RAD technologies [96, 31, 17]) typically irrelevant to the target platforms and implementations. The latter case in fact is sometimes unavoidable because a prototype is likely to include hardware and software functionality which are very difficult to capture in the same design model.

To simplify these concepts, let us assume we are to model an airplane system to study and explore its characteristics. A small wooden model of the airplane will precisely describe the “shape” of the airplane but does not describe its aerodynamic characteristics (e.g., test the model in a wind tunnel). We will need to build a metal airframe (assuming the airplane is be made of metal) to use it in the wind-tunnel to verify the aerodynamic characteristics. The two models have a different purpose and representation.

The first model is made of wood and this helps to rapidly create the airplane shape and explore how the different airplane parts, such as wings, tail, fuselage, cockpit, etc., are (relatively) fit together. The small size allows designers to have a compact view of the airplane, and the wood is a cheap means that allows rapid and repeated prototyping until we get the right model. Such a model is a throw-away prototype.

The second model however is a big real-sized airframe (not suitable for conceptual modeling as is the wooden model) and made of real, sophisticated material – so it’s suitable for real tests in the wind tunnel and also amenable to further implementation and building of a complete airplane.

Reflecting back on software systems, a synthesized design model of software that we target is analogous to the real airframe. However the common use of prototyping concepts in software is analogous to the wooden model of the airplane.

2.2 Running Example

The Engineered Safety Feature Actuation System (ESFAS) of a Nuclear Power Plant is intended to prevent or mitigate damage to the core and coolant subsystem of the plant on the occurrence of a fault such as a loss of coolant. A simplified specification of ESFAS [2] is commonly used as a case study in the computing literature. We will use the ESFAS system as a running example to illustrate the concepts described in this chapter, and later in other chapters too.

The ESFAS receives signals from sensors and determines whether or not predetermined set-points are being reached; if the system logic determines that they are reached, actuation signals (safety injection) are sent to safety feature components the function of which is to cope with the accident.

A manual block (push button) is provided in order to override the safety injection signal and to avoid actuation of the protection system during a normal start-up or cool down phase. A manual block is permitted if and only if the steam pressure is below a specified value (permissive). The manual block must be automatically reset by the system. A manual block is effective if and only if it is executed before the protection signal is present.

2.3 Interactions-Based Specifications

A generally accepted interpretation of a *scenario* is a sequence of interactions steps between the computer system and the outside environment [27, 28]. A scenario partially describes the computer system behavior because it specifies its reactions to the environment's stimuli as far as this specific scenario is concerned. A set of scenarios will be considered as completely specifying the computer system behavior only if they specify all possible environment stimuli and possible combinations. In practice, however, such complete requirements are not readily available, particularly at early development stages. A fundamental reason is that, typically, scenarios are provided by different stakeholders with different viewpoints and needs [45].

Several graphical description techniques for specifying scenarios interactions have emerged over the past decade. In this section we discuss several of these dialects to get an impression of the existing syntax, their application domains, as well as their semantic foundations.

We start our discussion with Message Sequence Charts (MSC-96) [69, 70], whose origin is the specification of interaction scenarios in the telecommunications industry sector. The MSC-96 specifications have rich syntax and corresponding formal semantics definition, which cover not only finite interaction scenarios, but also infinite behavior.

Moreover, our interest in this notation is triggered by its means for composing elementary

scenario specifications into compound scenarios – going beyond what most other notations cover – which is an important synthesis-friendly feature. Later, we discuss in Chapter 5 the means for structuring of scenario specifications, where we propose our novel structuring framework.

Therefore, we give a thorough introduction to MSC-96’s syntax, and let it serve as the reference notation during the discussion of the other MSC dialects. We also briefly mention the extensions introduced by MSC 2000 [71], the successor to MSC-96 [69, 70].

One of the main application domains for MSCs is the specification of communications protocols. However, the increasing interest in interaction scenarios and use cases in object-oriented analysis and design have spawned several dedicated MSC dialects. These dialects integrate special syntax for method calls and control flow constructs, but typically cover finite interaction patterns only. In the following discussions, we use Object Message Sequence Charts (OMSC) [26] and the UML Sequence Diagrams [103] as representatives of this category of MSC-based dialects. This will provide the necessary basis for the discussion in Chapter 5, where we will revisit the recent UML’s approach for specifying Interactions Specifications.

We also discuss the MSC-based dialects of Life Sequence Charts (LSC) [38], Hybrid Sequence Charts (HySC) [56] and Triggered Message Sequence Charts (TMSC) [123]. We selected them as representatives of other dialects that we do not survey here, in order to keep the discussion focused. These dialects are not just variants of MSC’s syntactic constructs. LSC emphasizes on logical modality concepts such as *universality* and *existentiality* drawn from Modal Logic [30]. Life Sequence Charts and Hybrid Sequence Charts assign a formal semantics to individual component states as part of interaction patterns. Hybrid Sequence Charts also contribute syntax and semantics for preemption specifications to the MSC notation. The reader should note that we deliberately deferred the dialects and notations used in scenario-composition. We spare Chapter 5 to discuss these techniques within the context of structuring the scenario specifications where we demonstrate our novel approach to structuring scenarios. In the following sections we introduce all of these notations in more detail. We have tried to keep an example-oriented style of presentation. In this way, we not only convey the features of the different MSC dialects, but also give an impression of their “look-and-feel.”

We have deliberately refrained from judging the design choices made, or syntactic and semantic constraints imposed by, the authors of the respective description technique. The purpose of this chapter is to expose the syntactic features of several MSC dialects, and their potential as synthesis-friendly specifications.

2.3.1 MSC-96

Message Sequence Chart a description technique for component interaction recommended by ITU [69, 70, 71, 72]. The version of this recommendation we cover here is called MSC-96; it introduces both a graphical and a textual syntax for MSCs. For the purposes of this section we concentrate on the graphical syntax.

The intended use of MSCs is to provide an “overview specification of the communication behavior for real time systems, in particular telecommunication switching systems,” (cf. [69, 70]). In fact, MSCs have their roots in the development of telecommunication systems using the Specification and Description Language (SDL [42]). We refer the reader to [97, 119] for a detailed overview of the history of MSC-96.

The basic assumption underlying the use of MSC-96 is that the system under development consists of a set of components, communicating by means of asynchronous message passing. There is no notion of a global clock; the components (or instances, as [69] and [70] call them) operate time-asynchronously.

Compared to other MSC notations (covered in Sections 2.3.2-2.3.4), MSC-96 is a rather rich language; it provides constructs for specifications ranging from simple interaction patterns to complete component behavior. We introduce these constructs along the following structure. In Section 2.3.1.1 we discuss the basic notational elements of MSC-96: instance axes, message arrows, environment frame, actions, and conditions. These constructs suffice for the specification of simple interaction patterns. MSC-96 also enables the capturing of alternative, repeated, and parallel interaction patterns. We deal with other constructs for structuring and composing MSC specifications, as well as with reference expressions (a substitution mechanism), gates (a means for splitting information among several MSCs), and High Level MSCs (a hierarchic notation for MSC composition) in a separate discussion in Chapter 5.

We refer the reader to [70, 69, 119] for a detailed presentation of MSC-96’s syntax and semantics, including all syntactic options we avoided here for reasons of brevity.

2.3.1.1 Basic MSC Notation

We start with the most basic MSC constituents: instance axes and message arrows. These two ingredients reappear in almost all graphical notations for component interaction. MSC-96 complements them by symbols for specifying component conditions and component actions. In the following paragraphs we discuss all of these concepts, as well as the ones MSC-96 provides for defining the ordering of events within MSCs.

Figure 2.1: Example of Basic MSC diagram

Instances and Messages: Figure 2.1 shows an elementary MSC in graphical form. This scenario is drawn from the Steam Boiler case study that we investigate in Chapter 5. It displays a sequence of interactions among three instances that are represented by vertical axes.

An *instance* refers to an (instantiated) object in the system, and pictorially represented by a head-box known as *instance head symbol* as in the MSC diagram; for example, the SafetyHandler in 2.1 is an instance.

An axis starts with a non-filled rectangle and ends with a filled rectangle. The non-filled rectangle is the instance head symbol, the filled rectangle is the “instance end symbol.” Labels at the top of each axis indicate the corresponding instance’s name. In Figure 2.1, the instance names are Operator, ESFAS, and Pressurizer. Arrows, directed from the sending to the receiving instance, denote communication. In Figure 2.1, examples of message labels include PowerON, PressureLow, SafetyOn, etc. The frame around the instances is part of the MSC; it represents the environment. The name of the MSC, given after the boldface keyword **msc**, serves as an identifier for referencing the entire MSC. In this case, the name of the MSC is BoilerStartup.

The semantics of MSCs is formally defined by a process-algebraic model in [70]. Intuitively, the meaning of an MSC is the sequence of messages obtained by reading the MSC from top to bottom. Because – according to the documents defining the standard [70, 69] – there is no global clock, there may exist an individual time scale for each instance depicted in an MSC. Moreover, communication happens asynchronously; there may be a delay between the sending and the receipt of a message. Therefore, for any message the authors of the semantics document [70] distinguish between two events: the sending of the message, and its receipt. If the name of a message is m , we denote the corresponding send and receive event by $s.m$, and $r.m$, respectively. As a consequence of the absence of a global clock the events on different instance axes are not ordered per se. The ordering is established by several rules in the MSC-96 semantics; we mention these rules informally along with the introduction of the respective syntax elements.

The following four restrictions, imposed by the MSC-96 standard, induce a partial order on the events occurring within an MSC:

1. every send event precedes its corresponding receive event;
2. on any location on an instance axis at most one send event may occur;
3. at most one receive event may be at the same location as a send event on the same axis; the receive event precedes the send event;

4. the events on a single axis are totally ordered according to their occurrence from top to bottom.

The semantics of an MSC, according to [70], is the set of all sequences of events that correspond to the messages depicted in the MSC and obey these rules. MSC-96 offers two constructs to relax the ordering imposed by these restrictions: co-regions and generalized orderings. We will discuss these advanced concepts later in this section.

Discussion:

Message ordering is useful and actually a synthesis friendly construct. This ordering should prescribe a specific path (i.e., a series of states and their transition events) in the synthesized state-machine model. So, synthesis algorithms should consider this ordering as a constraint in the input specifications. Note that this applies to the state-machines generated individually from the input MSCs, as well as the overall machine resulting from the merge of these individual machines. The point is that a prescribed path in a state-machine (generated from individual MSCs) might emerge when all individual state-machines are integrated into a single overall machine of the system.

To conclude, as far as the current MSC standard is concerned, a message ordering in an MSC should be held as a constraint across the system. More advanced techniques will be needed in case these ordering need to be associated with particular conditions. Our scenarios-structuring approach (Chapter 5) organizes scenarios such that conditional ordering can be specified.

Environment Frame

The environment frame of an MSC serves as a representative for the origin or destination of arrows whose source or destination, respectively, is outside the scope of the MSC. This allows us, for instance, to leave the concrete sender or receiver of a message unspecified. To denote message exchange with the environment we position the corresponding arrow's head or tail at the frame. As an example, consider Figure 2.2. Here, instance ESFAS exchanges messages PowerOn, PressureLow, SafetyOn, PressureNormal and SafetyOff with its environment.

Besides modeling communication with (possibly) anonymous environment instances, the environment frame also allows for breaking up a single MSC into two, such that the source and the

Figure 2.2: Example of the environment frame in MSC

destination of arrows can end up in different MSCs. We will come back to this purpose of the environment frame later in this section, in connection with MSC-96's gate concept.

Conditions

MSC-96 offers condition symbols as a means for indicating that one or more instances fulfill a certain condition before or after they participate in an interaction sequence. Graphically, a condition is a labeled angular box placed on the instance axes of the components fulfilling the condition. As an example for an MSC with conditions, consider the MSC in Figure 2.3.

MSC-96 distinguishes three kinds of conditions, according to how many instances of an MSC they cover. Local conditions cover exactly one instance axis, nonlocal conditions cover more than one instance, but not necessarily all of them, and global conditions cover all instances. The intuition behind nonlocal and global conditions is that the covered instances all fulfill the stated condition.

Semantically, conditions serve only one purpose in MSC-96: they are the basis for composing MSCs within High Level MSCs (see Chapter 5). Besides that, conditions do not contribute to the meaning of an MSC [70]. Although MSC-96 does not assign much meaning to conditions, they have many useful applications. For instance, the developer may use conditions to mark phases of a communication protocol within an MSC. In this way, conditions can enhance the readability of an MSC specification. Another application is expressing information on the control and data state of an instance by means of appropriately labeled conditions.

msc BoilerStartup-cond

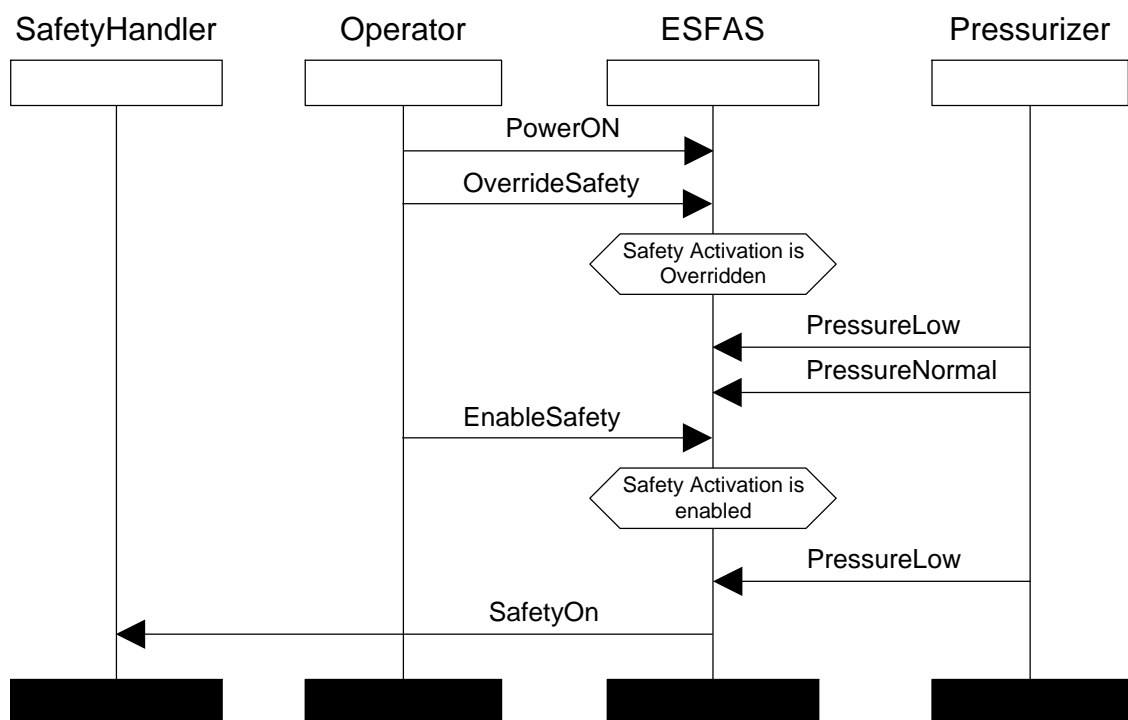


Figure 2.3: Example of Conditions and Actions in MSC

Discussion:

Conditions on an instance in the MSC represent the (observed) states in which the instance exists during the execution sequence of the MSC. A condition s on an instance O , coupled with its preceding and succeeding m_pre and m_post messages (on the instance O) represent the basic unit of translating a scenario to an FSM: the condition s is a state in the FSM and the messages m_pre and m_post are its incoming and outgoing transitions. We will come back to this in details in Chapter 3 and Chapter 5.

For a scenario to be synthesis-friendly:

- 1. Each instance (or the instance representing the system under development) must be annotated with conditions.*
- 2. The conditions can be either labeled conditions or detailed state information. The former should translate to automata models such as Labeled Transition Systems (c.f., [78]) and the later should translate to state-based automata models such as Kripke Structures [82]. We discuss the pros and cons of each type of model in Chapter 3 where we justify our selection of the appropriate output of our synthesis process in Chapter 6.*
- 3. The conditions should not be over-specified. Coding the conditions as complete state information (i.e., valuation of all state variables) has its pros and cons. One down side of providing complete state information is that it may constrain the integration of the specifications together in one model. Furthermore, conflicts may arise between the scenarios if the conditions have not been consistently specified. We discuss this in more detail in Chapter 6. In general, scenario specifications should defer complete state information in conditions to the design (synthesis) phase unless such complete information is itself a requirement. In turn, synthesis processes should be smart enough to provide the missing state information such that it avoids conflicts.*
- 4. Pre- and Post-conditions should be specified before and after each message, respectively. This allows the synthesis process to deterministically identify the states in the target FSM translated from the scenario.*
- 5. Also, a scenario will be synthesis-friendly if the messages themselves are coded as conditions, instead of just events drawn from some alphabet. A message condition translates to the*

trigger that causes the transition between the two states that correspond to the pre- and post-conditions. As we will see in Chapter 6, this has the advantage of allowing the synthesized model to be more convenient for analysis using Theorem Provers (c.f., PVS [121]) and model checking techniques [33].

6. *Finally, the scenario flow implicitly assumes that consecutive conditions on an instance correspond to contiguous states in the system behavior. This assumption is a two-edged sword. On the one hand, it might cause over-specification of the scenario where conditions are “strongly” coded via complete valuation of state variables. As mentioned in point (3), this potentially constrains (and sometimes prohibits [134, 133]) the integration of the individual scenarios’ behavior models during the synthesis process. On the other hand, with “weaker” conditions, ambiguities (and sometimes, inconsistencies) may arise during the synthesis process and it requires a designer’s intervention in order to be resolved. Practically, little knowledge is available at early stages of development and scenarios with weak conditions are sometimes unavoidable in the specifications. Another supportive argument is that deferring the decisions is more convenient for wider and more flexible solution selection during the (synthesis-driven) design task.*

Actions To specify that an instance performs some local activity (such as state changes through assignments), MSC-96 provides the concept of actions. Their graphical representation is a labeled rectangle attached to the instance performing the action. Semantically, an action represents a local event of its corresponding instance. Action events contribute to the total event ordering along an instance axis, and, hence, indirectly to the partial ordering of all events occurring in an MSC.

Discussion:

From a synthesis viewpoint, actions in MSCs are no different from normal events other than that fact that they are generated by the same instance. Accordingly, actions should have pre- and post-conditions as well as the action’s condition. Typically, the state-change due to a local action is known at specification time, e.g., the expiration of a timer variable, and no risk of over-specification. It is relevant to note here that during the refinement phase of a synthesis

process, designers introduce more behaviors by adding (internal) state variables (e.g., timer variables, index counters, etc.) and the associated states and transitions to the model under synthesis. Typically this is done due to further elicitation or elaboration of behavior, or as a side product of resolving some inconsistency in specifications. This is one of the benefits of a smart synthesis procedure: discovering gaps in the behavior space and helping to elicit or introduce behaviors addressing these behavior gaps. In brief, actions are important ingredient of the system behavior model, whether they are specified within scenarios or added during the synthesis process.

Event Ordering Mechanisms

The ordering of events in an MSC is defined by the rules we have given above. There are situations, however, where these rules are either too strong, or too weak to express the desired interaction sequences succinctly with MSCs. Consider, for example, the MSC of Figure 2.1. Its semantics state that the messages SafetyON and SafetyOFF messages arrive at the instance SafetyHandler in the order drawn in figure 2.1, however this MSC alone cannot describe situations in which SafetyON arrives at the SafetyHandler instance after SafetyOFF. MSC-96 provides coregions for weakening, and general orderings for strengthening, the standard event ordering.

Coregions

A coregion, whose graphical representation is a vertical dashed line delimited by short horizontal lines, is part of an instance axis. All events located in a coregion are unordered. Figure 2.4 illustrates an example scenario with the coregion construct. In this figure, the two messages PressureLow and PressureNormal are located in a coregion, so these two messages are unordered.

General Orderings

A general ordering establishes a precedence between two events in an MSC that would be unordered otherwise. It is represented graphically by a (possibly bent) line between the two events under consideration. Attached to the line is an arrow head that points from the event occurring first

msc BoilerStartup-coreg

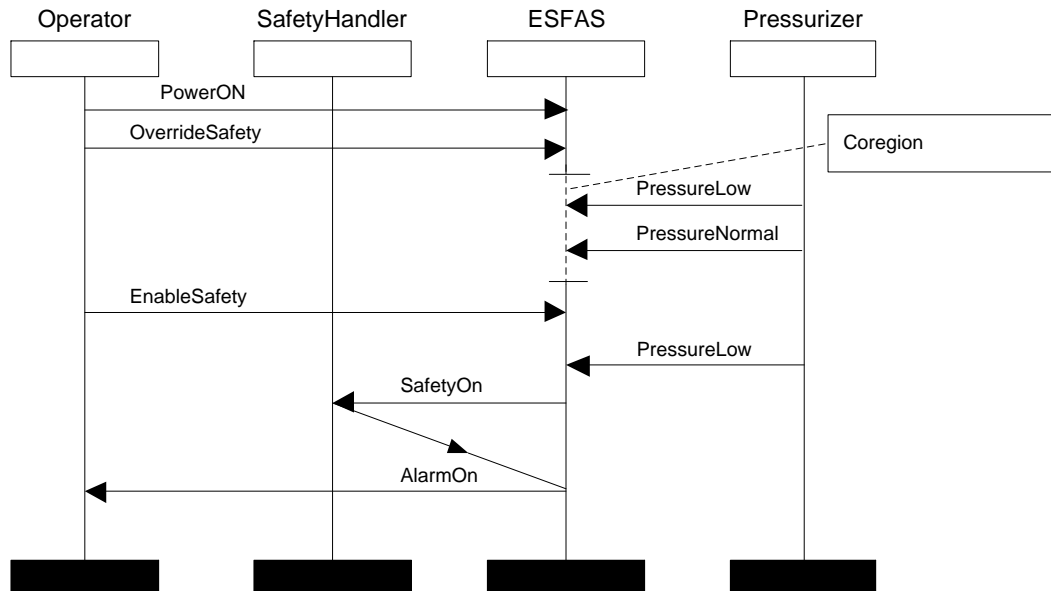


Figure 2.4: Example of coregions and general orderings in MSC

to the one occurring second. To distinguish general ordering symbols from message arrows, the former's arrow head's position must not be at the end of the line connecting the two events.

Figure 2.4 also shows an example of this construct. In this Figure, the two events of 'receive of SafetyOn message' and 'send of AlarmOn message' could happen in either order. The General Ordering construct can be used to prescribe precedence on these two events as depicted in Figure 2.4.

We may use general orderings also for ordering events within coregions. We refer the reader to [70] for more details.

Discussion:

Although the scenario flow implies a default ordering of events, this flow can be emphasized by imposing a General Ordering between events, and can be deferred to a later refinement stage, as discussed above. As with the case of an instance's conditions, strengthening the ordering of events might over-specify the requirements, if not well thought out. An important task in the synthesis process is to integrate the behavior models translated from the individual scenarios, and having strict event ordering might prohibit such integration. The MSCs semantics does not explain if the

event ordering disallows other intermediate events to be introduced, which is expected during the integration phase in the behavior synthesis process. For example, when two scenarios are integrated (e.g., interleaved) together, two ordered events in one scenario might not execute consecutively if an intermediate event from the other scenario is interleaved with them. The real problem arises if the intermediate event negates the first ordered-event, and so the second ordered-event will actually execute out of the specified order. Weak ordering of events allows the synthesis process the flexibility (but possibly the ambiguity) of ordering those events. The upside of weak ordering of events is the flexibility offered in the synthesis process to integrate the system behavior model, and the obvious downside is the ambiguity potentially arising during the synthesis process that might require domain experts to resolve.

Practically speaking, in our opinion, imposing event ordering at an early stage of development is not recommended and its expressiveness benefit does not outbalance its potential problems during synthesis. As an alternative, event ordering can be expressed as constraints that can be assured, by the synthesis process, to be satisfied in the generated behavior model.

2.3.1.2 Programmatic Constructs of MSC-96

The syntactic and semantic elements of MSC-96 introduced so far already suffice to model simple interaction sequences. In many cases, however, we are interested not only in depicting one interaction sequence, but also in alternative sequences or repeating the same sequence within the same MSC. We may, of course, draw individual MSCs for each possible combination of message exchanges, and accompany this with an explanation of how these different pictures relate. However, for non-trivial systems, this would lead to very large numbers of MSCs.

To overcome these cases, the MSC-96 standard provides several features for constructing more complex MSC interactions. We categorize these features into two groups: programmatic features and composition features.

The programmatic feature in MSC-96, known as *Inline Expressions*, supports the specification of looping, conditional-selection between alternative sequences (can be thought of as “if-else”) and parallel interactions (which can be thought of as spawning a thread for each parallel sequence). We detail these features in this chapter.

The composition and structuring features include:

- *References*: that allow for reuse of interaction patterns among MSCs,
- *High Level MSC*: to provide flowcharting-like constructs to relate sets of MSCs in similar fashion as the programmatic features do within a single MSC,
- and *Gates*: to facilitate the decomposition of MSCs.

In the rest of this section we detail the MSC-96 programmatic constructs within a single MSC. We defer the discussion of inter-MSC composition features to Chapter 5 where we survey other relevant scenario composition approaches and propose our novel framework for structuring scenarios.

Inline Expressions MSC-96 provides the notion of inline expressions to represent alternative, optional, parallel, and repeated parts within an MSC. The graphical syntax of an inline expression is a rectangle whose upper left corner indicates the expression's type. Every inline expression covers at least one instance, and makes a statement about the events within the rectangle.

Alternative Inline Expressions: Alternative constructs include the following:

1. The **alt** construct: this allows to specify two execution branches in the scenario flow. There can be more than two alternatives.
2. The **opt** construct: this allows to specify the case of an **alt** construct where one of the alternative flows is empty (i.e., has no messages).
3. The **par** construct: this allows to specify parallel behaviors in scenarios. Basically, the **par** construct group message that are independent (i.e., they have no temporal relationship or shared data) that can be executed in parallel.
4. Finally the **Loop** construct: implements the standard looping features that are widely common in programming paradigms. In MSC, the Loop expression simplifies the diagrammatic depiction of repeated interaction patterns. Below is brief syntactic description of this construct:

loop $\langle l, u \rangle$: let $l, u \in \mathbb{N}$. If $l \leq u$, then the interaction sequence may occur at least l and at most u times. If $u > l$, then the semantics of the loop-expression is equivalent to that of the empty MSC. If $l \in \mathbb{N}$, and $u = \infty$ then the number of occurrences of the interaction sequence is bounded only from below by l .

loop $\langle l \rangle$: if we have $l \in \mathbb{N}$, this is a notational shortcut for **loop** $\langle l, l \rangle$; which specifies that the interaction sequence occurs exactly l times. If $l = \infty$, then the interaction sequence occurs infinitely often.

loop : this is a notational shortcut for **loop** $\langle l, \infty \rangle$; the interaction sequence may occur an arbitrary number of times.

The Inline Expressions are powerful syntactic constructs, similar to the programming constructs in general purpose languages. They are introduced in MSC as part of its arsenal of graphical notations, and aligned with its graphical representation to improve expressiveness. They are useful to build compact artefacts of specifications and allow the specifier to capture interaction patterns in the familiar programmatic techniques that all software engineers are acquainted with.

However, from a modeling viewpoint, these constructs are not suitable as techniques for writing early specifications. This is basic motivation for our approach here for structuring (basic) scenarios into more compound models (as we will see in Chapter 4 and 5) using constructs more abstract than programming techniques.

Discussion:

Inline Expressions within an MSC are consumable by the synthesis processes, their use of standard constructs (e.g., loops and alternatives) facilitates straightforward translation to automata-based behavior models, and their compactness minimizes redundancies in the translated models.

Inline Expressions, however, have a higher risk of over-specification than is the case with other expressive notations in MSCs. The programmatic nature of Inline Expressions tempts the specifier to dig in more and over-specify the scenario (even though this is useful for eliciting more requirements). Moreover, the compound nature of Inline Expressions can easily complicate an MSC, thus burying its original characteristic of simplicity and intuitiveness. In brief, despite their benefits, care and discipline must be considered when specifying such programmatic constructs in scenarios. This is one of the important issues we address in this thesis, as we will discuss in Chapter 5. We postulate that, principally, the complexity of scenario specifications can be managed if we structure the specifications within the behavior space. This benefit feeds in directly to the synthesis process, we as will see in Chapter 6, and allows better generation of output models without hidden gaps in the behavior space.

2.3.2 LSC

Damm and Harel [38] have introduced Live Sequence Charts (LSCs) as an extension of the ITU's MSC-96 standard for plain (basic) MSCs. The major addition here is integrating the specification of liveness properties into the MSC notation. To that end the authors relate LSC specifications to system runs. A system run, in their approach, is an infinite sequence of snapshots, where a snapshot consists of the set of current events (being either synchronous or asynchronous *sends* or *receives* between components or between a component and the environment), and an assignment of values to all variables of the system.

The authors in [38] associate liveness with four syntactic elements of MSCs:

- plain MSCs as a whole,
- locations (segments on an individual instance axis),
- messages,
- and conditions.

We can specify the occurrence of each of these syntactic elements as either mandatory or provisional (but not both) in LSCs. In the following paragraphs we briefly address each of these items in turn.

Plain MSCs: in [38], Damm and Harel associate a mode with each plain MSC. A mode can be universal (i.e., a mandatory MSC) or existential (i.e., a provisional MSC). Any system run must satisfy a universal LSC, whereas an existential LSC requires only at least one such run to exist. Syntactically, the frame around an LSC determines whether the LSC is a universal (solid-line frame) or an existential (dashed-line frame) one.

Location: A dashed line segment on an instance axis denotes that, during a run of the system, the instance under consideration need not move beyond this line segment. A solid line segment indicates that during a system run the instance must move beyond this line segment. Intuitively, the distinction between these two cases allows the developer to specify local progress requirements in the global context of an interaction sequence.

Message: a dashed arrow denotes that the corresponding message, if sent, may or may not arrive at its destination. Solid arrows indicate that a sent message must arrive. In addition to asynchronous message exchanges (indicated by open ended arrowheads) known from MSC-96, the authors allow synchronous message exchange (indicated by solid arrowheads).

Condition: the authors of [38] associate state predicates with conditions occurring in LSCs. If, during a run of the system, execution reaches a provisional condition evaluating to false, then the LSC allows arbitrary behavior from this point onward. If, during a run of the system, execution reaches a mandatory condition evaluating to false, the modeled system halts. If the condition holds in either case then execution simply progresses beyond the condition. Provisional conditions appear syntactically as MSC-96 condition symbols with dashed outlines; the graphical representation of mandatory conditions are MSC-96 condition symbols with regular outlines.

The authors hint at, but do not elaborate on, the representation of repetition constructs; similarly, they only briefly mention “forbidden” scenarios, i.e., scenarios that must not occur in a system run. They also introduce the concept of “subcharts”, which are LSCs occurring within other LSCs. The idea is that subcharts, in which a provisional condition does not hold, terminate, and execution continues in the “parent” LSC outside the subchart; one application, therefore, of subcharts together with provisional conditions is to specify preemption in LSCs.

2.3.3 OMSC

Object Message Sequence Charts (OMSCs) were introduced by Frank et al. [26] to describe interaction patterns in object-oriented software architectures. The basis of OMSCs are MSCs (see Section 2.3.1). Because of their intended application, however, the syntactic elements of OMSCs differ significantly from those proposed in the MSC-96 standard. Moreover, OMSCs do not provide any formal semantics; the authors in [26] describe their use for explanatory purposes, but neither their semantic foundation, or their integration into the development process. Because the method call, and often also the yielding of control between caller and callee, is of major concern in object-oriented designs, the developers of the OMSC notation offer specific syntactic support for these modeling tasks.

Our focus here will be on the distinguishing features of OMSC compared to MSC-96 and their relation with synthesis processes. We refer the reader to [26] for detailed presentation of OMSC notations.

OMSCs provide the following modeling elements:

- **labeled axes**, representing part of the existence of an object (this corresponds to instance axes in MSCs);
- **labeled arrows**, indicating message exchange, method calls and returns (this corresponds to the message arrows within MSCs, although MSC-96 does not offer **synchronous** and return

arrows);

- **object activities**, denoting phases where an object is active, i.e., where it executes the body of a method, function, or procedure (there is no corresponding concept in MSC-96);
- **object creation arrows**, denoting the point from which on an object exists (this corresponds to the createline of MSCs);
- **object deletion symbols**, denoting the end of an object's existence (this corresponds to instance stop in MSCs);
- **process boundary symbols**, denoting what set of objects belongs to what process of the system under design (there is no corresponding concept in MSC-96);

Elements from MSC-96 absent from OMSCs are the environment frame, conditions and actions, inline and reference expressions, coregions, general orderings, gates, lost and found messages, timers, and hierarchy. In particular, the lack of repetition and referencing constructs restricts the use of OMSCs to representing interaction scenarios of rather limited size. The major difference between MSC-96 and OMSCs is that OMSCs introduce the notion of control flow into the specification of interaction descriptions. This has gained them significant popularity especially among software engineers who use scenarios to describe sequences of method calls with corresponding returns. Because OMSCs support depicting control flow, method calls, and returns, the authors of the UML [120, 103] selected them as the basis for Sequence Diagrams. The lack of a formal semantics for OMSCs however hinders their application in rigorous modeling environments.

Discussion:

Control flow and synchronization between objects do not affect the synthesis of each object. However, specifying control flow and synchronization manually is a fertile environment for conflicting and inconsistent specifications. These problem will surface only during the synthesis process when we try to merge the automata models of the same object generated from different OMSCs. So, in conclusion, while control flow facilitates decisions when generating global automaton of the individual component (or object) automata, inconsistencies might arise in the process of generating the individual automata of the components themselves.

OMSC constructs are highly biased by the control flow of programming languages. At early stages of development, particularly in the absence of a decomposition of the system, we have only a set of variables with some domain theory and typically no decision on methods or APIs have been made yet.

2.3.4 HySC

Radu et al. [56] use the syntax of MSC-96 for the specification of interaction behavior in hybrid systems. The corresponding semantics differs significantly from the one of MSC-96; the semantics of Hybrid Sequence Charts (HySCs) bases on a shared variable communication model for clock-synchronously operating components.

We mention this MSC dialect for two reasons. First, it assigns a formal semantics to condition symbols (as predicates over discrete and continuous component variables), and provides access to a quantitative notion of time in MSCs (via appropriate differential equations). Second, it makes the concept of preemption syntactically and semantically accessible to MSC specifications.

2.4 Other Forms of Interaction-Based Specifications

In the previous section we went described various dialects of scenario-like specifications. There are other approaches that specify the system behaviors in terms of interactions between the system under design and its environment, but do not necessarily use a scenario-like format. These approaches are of lesser relevance to this thesis, based on their lesser utility to a non-technical audience (the client).

The classical example of such interaction-based specifications are the *formal languages* [67] and their various extensions. Also event-based approaches, such as CSP and CCS [66, 101] are more elaborate dialects of formal language specifications.

Desharnais et al. [39] applied *relational methods* [66, 101] to formalize the relationship between the system under development and its environment. This relationship is captured in so-called *Sequential Scenarios*. The definition of these scenarios captures the functional aspects of the interactions between a computer-based system which is the system under development, and an unconstrained system that represents the environment. The key contribution of this approach is the use of the relational property known as *Demonic Meet* [39] to define the integration (i.e., the merge) of these scenarios.

We refer the reader to the the relevant computing literature for comprehensive surveys (c.f., [119]).

2.5 Chapter Summary

In this chapter, we have presented a concise overview of interaction-based specification approaches, and we have focused more on scenario-like forms of those approaches. We went through the basic features of different interaction-based specifications dialects, such as MSC, LSC, and we gave illustrations where necessary. Also, we provided analytical discussion of specification constructs commonly used in scenarios, and the feasibility of these constructs as input to synthesis processes that generate state machine models from scenario specifications. We highlighted the strengths and weaknesses of each type of construct.

Not all of the scenario constructs are synthesis-friendly. For example, the Event Ordering mechanism in scenarios may cause indeterminism in the synthesized models. Fortunately, most of the other constructs, such as Conditions, are very useful input to the synthesis process. In this thesis, we take advantage of these synthesis-friendly and augment them with *mode-based specifications* to (semi-)automatically generate an integrated behavior model from a given scenarios.

Chapter 3

Software Behavior Model Synthesis

This chapter surveys and evaluates the state-of-the-art in automated synthesis of behavior models, and analyzes the associated techniques.

In Chapter 2 we discussed the specification of the system’s behavior requirements that are input to the synthesis process, and we discussed the synthesis-friendly criteria for such specifications in order to be usable by the synthesis processes. In this chapter we rather focus on the output of synthesis. We discuss various forms and features of synthesizable “solution” models, output from the synthesis process, and we survey existing techniques for constructing those models via the synthesis processes.

The discussion in this chapter provides an overview of the current theory and practice of applying the concept of synthesis in software systems engineering. Additionally, we conduct an objective comparison of current synthesis approaches, which help to underline the major issues by which the synthesis processes are challenged, and thus preparing for the discussion in Chapter 5 of our technique to structure scenario-based specifications, which we employ primarily to address many issues we discuss here.

3.1 Introduction

Software design methods offer a limited support for modeling the system behavior and behavior composition, compared to established structural modeling frameworks, such as object-orientation [26, 103] and the original work of modular design [105]. Typically, system behavior modeling follows the phase of structural decomposition. For example, it is not uncommon for designers to specify the automaton of each individual component and then use automata-composition techniques, e.g., Communicating Automata [24], to construct an overall system behavioral model. An interaction

scenario between the system's components is another example of a system behavioral model that follows the system's structural decomposition, where interactions between the individual components are based on their relationships in the structural model (e.g., require/provide interfaces of each component). Note that a scenario is a behavior specification model and can not be a "solution" model – which we seek in this chapter – that involves design decisions and details drawn from the solution space.

The methods and techniques we contribute in this thesis are concerned with the behavior design of the system as a black-black, without necessarily breaking it down, structurally, into objects. The major challenge facing behavior modeling methods is the complexity of building the models in the first place. Even though structural modeling and methods help to simplify the system by decomposing it into a set of manageable components, modeling of the overall system behavior, as well as the behavior of the individual components, is still a laborious task. This is clearly evidenced by the numerous dialects (as we discussed in Chapter 2) proposed to model these interactions and the sophistication required to support concurrency and real-time aspects.

Considerable research work is done to automate the complex task of building software behavior models from given requirements. Most approaches accept as input a set of requirements specifications in the form of interactions specifications or other forms of declarative specifications, and attempt to (semi)automatically synthesize a design model, mostly an automata-based model. Such automata-based models exhibit the behavior specified in the input specifications, in addition to behavior details introduced during the synthesis process itself.

In this chapter, firstly we establish a detailed background on concepts related to synthesis; secondly we extensively survey and analyze the different approaches of synthesis from interactive specifications (mainly scenario-based), and finally we provide our evaluation and identify the research gap. This thesis is particularly concerned with synthesis from interactive-specifications, so we omit other techniques used to synthesize behavior models from non-scenario intuitive specifications such as goal-based and property-based specifications [77, 86, 90]. In Section 3.2, we look at the basic concepts of modeling in software engineering and the synthesis of these models with highlights from other areas of computer engineering. This provides a background on fundamental concepts discussed in this chapter. Section 3.3 surveys current synthesis approaches and identifies the relevant strengths and weaknesses of each. Finally we provide a chapter summary and outlook in Section 3.4.

3.2 Background

Specification and modeling of software systems is a broad topic that has received extensive attention from both computer scientists and engineers. Different terminologies have developed in academia and industry although they refer to and address the same subject or artefact. This sometimes led to the 'reinvention' of some of the basic 'wheels' in computer science, in general, and software engineering in particular.

To avoid this confusion, we will outline some basic concepts of system behavior and system models. Whenever there are several definitions of the same concept, we will refer to the specific one adopted in this thesis.

Moreover, we provide the historical background of Software Synthesis – the subject matter of this thesis – and examine how it has been applied to solve several software engineering problems, and we briefly examine related uses in digital hardware synthesis. We also refer to the classic concept of prototyping that we discussed briefly in Chapter 2, and we clarify the gray area between it and the concept of synthesis. Finally, we take a quick look at different automata-based behavior models – the output of the synthesis process – and we discuss the various automata dialects, focusing on the specific form of automata that we employ in our synthesis process.

3.2.1 Models *and* Software Behavior Models

System behavior is a well understood concept in almost all engineering disciplines. The methods and tools used in established engineering disciplines are mature enough to allow engineers to use them to capture system behavior. In software-built systems, however, there is little or no convergence on the effective methods and techniques to support the process of capturing the behavior via some artefact.

The Modeling Paradigm. *System Modeling* and *Model-Driven Engineering* [46, 111] concepts have promoted the development of models that capture system's properties of interest. A *model* of the system is an abstract (and hence, a simplified) depiction of the system or the product under development [109]. By definition, a system model is intended to represent some (and typically not all) of a system's properties. However, a model should not have properties that do not exist in the real system. Such models are known as a high fidelity models [126] of the system. Consider the example of building a plastic model of the body-frame of an aircraft. The model is purpose-built to reflect exactly the geometrical properties of the aircraft's frame, but not its material properties.

In software engineering, a plethora of approaches have contributed various modeling notations and techniques. These type of these notation range from the extreme of pure theoretical notation that

addresses certain aspects of the system (c.f., [66, 8, 25, 101]) to the other extreme of informal approaches [103, 47, 62] (despite the considerable efforts to formalize them). Other modeling paradigms have emerged in industry as *de facto* models. Examples of these latter models are SIMULINK MATLAB, www.mathworks.com, that are widely adopted in real world development.

Automata Models. Despite the wide range of existing specifications techniques, standard automata-based models are still the most commonly accepted models of software behavior. However, due to the lack of scalability of flat automata models, many approaches, led by Statecharts [59], attempted to introduce hierarchical structures into flat Finite State Machines (FSM) in order to expressively allow more compact representation of larger and complex systems, but this has come at a price. In particular, the authors of Statecharts [61] have not just introduced a simple hierarchy to state machines, but also provided other features such as super-state, parallels states, event broadcast, suspension, etc. that (in the light of our previous discussion) could easily introduce ambiguity. This is evidenced by the plethora of proposed extensions (including the Statecharts authors themselves, [61]) which attempted to add semantics for Statecharts (c.f., famous survey of Statecharts dialects [143]).

In this thesis, we opted for a different approach to state-space abstraction. We build on the classic idea of *Modes* that originate from hybrid systems [93, 139] to construct an early, abstract automata-based model capable of reflecting several views of the system behavior. *Modes* first appeared in software engineering in Heninger et. al. [63] to simplify the formal Functions Specifications [63]. We elaborate on the notion of *Modes* in Chapter 4 and use it to simplify and improve on the elicitation of scenario-based specifications.

In terms of the basic state-machine models (that we abstract via *Modes*), we assume simple FSM models as output of our synthesis process (Chapter 6). More specifically, we use *Kripke Structures* [82] that are pervasively used in Model-Checking techniques [33] due to their simplicity and clarity in other specialized automata models. It worth mentioning here that there are alternative state-machine models that we could have used, such as *Labeled Transition Systems (LTS)* [78]. However, we decided to use state-based models because they can consume specification types based on domain-variables, which we assume as the input specifications to our synthesis process. In Section 3.3 we survey the different models used by other synthesis approaches and, when necessary, we justify our use of Kripke Structures.

3.2.2 Prototyping and Synthesis

The use of Model-Driven Software Engineering [46, 47, 62, 103, 111] has gained momentum in the past decade. This paradigm builds on previous technological advances developed around the classic concepts of Prototyping and Synthesis. The terms *Prototyping* and *Synthesis* are frequently used in the computing literature to describe some process of constructing a System Model (see previous section). In Chapter 2 we elaborated on the distinction between the two concepts and discussed the common grounds between them. In this section we focus more closely on the concept of *model synthesis*, and discuss its different interpretations and applications.

In the computing literature, there are three schools of approaching *model synthesis*, we summarize "these as follows:

1. In the field of formal specification languages and temporal logic, *model synthesis* is understood as “the problem of building an implementation that will preserve a specification against any ‘malevolent’ environment” [1, 115]. More specifically, it is a constructive proof that the specification is implementable so that it does not contain any conflicting requirements. This definition applies, for example, to the problem that Harel and Kugler tried to solve [58]. Note that this is different from *satisfiability*, which asks if there is a benevolent environment where a specific implementation can be deployed. An example of work that tried to assume “synthesis” as a *satisfiability* problem is [20]
2. In the field of *machine learning* and *induction algorithms*, *model synthesis* is understood as the problem of inducing a universal rule describing every acceptable behavior of the system, given a set of example behaviors. In this context, scenarios are assumed as examples of behavior of the system under development. A set of scenarios is thus a finite set of example computations. This universal rule is typically a regular language (or a transition system,[67]) that accepts all the specified strings of events (i.e., behavior examples). Mäkinen et al., [94] and Hsia et al., [68] are two instances of existing work from this school.
3. A third school views *the model synthesis* problem as a standard compilation process. Scenarios here are considered as a complete description of the system under development. This is especially useful if the system to be built is closed [80]. Then, for each object in the scenarios, state machines are built from the various scenarios and finally integrated (i.e., merged) to form the total behavior of the object. There are two remarks worth noting here:
 - (a) If the scenarios describe the interactions between the environment from one side and the

system (as a black-box) from the other side, then the integrated state machine build for the system object represents the total (observable) behavior of the system under development.

- (b) However, if the system (under development) is decomposed into a set of components, and the scenarios describe interactions between these components and/or the environment, then a problem might *implied behavior* [9, 135] that might emerge. These behaviors might be undesirable or violate system properties. When recomposing the full system from the individual components, these behaviors emerge due to some dependencies between the global system view and the local view of each component.

Our work lies in the third category. We assume the synthesis problem as a process of elaboration and refinement of specifications, sharing the same understanding as several existing techniques in software engineering literature[83, 132, 133, 134, 136, 137, 145, 146, 149]. We compile an elaborate, state-based system from given set of scenarios, and refine this system down to a flat Kripke structure using an iterative synthesis process (Chapter 6).

The problem of *implied behaviors* [9, 135] is the major problem identified in synthesis from scenario-based specifications. We tackle this problem using a preventive approach – as opposed to protective approaches (c.f., [135]) – where we structure the state-space by multi-partitioning it into *mode-classes* to uncover possible gaps prior to specifying scenarios.

A more challenging problem emerged in this category of *synthesis* when merging the behaviors (of the same component) in a single integrated model. Uchitel et al., [133, 134], reported difficulties when attempting to merge the state machines generated independently from each scenario of the same component in one state machine. They concluded that in some cases, a common model of (or a model that exhibits) all these behaviors does not exist. We tackle this problem also by a preventive approach such that we specify an abstract common model (the *mode-classes* in our approach) and specify the individual scenarios within this common model, and avoid their inconsistencies in the first place.

We contribute a novel algorithm (see Chapter 6) for merging these state machines independently generated from different scenarios of the same component. A basic assumption in our approach is that scenarios are complemented by *mode-based* specifications (see Chapter 4 for detail). *Mode-classes* help to (initially) elaborate the scenario-based requirements and as a result they support requirements development at early stages of development (see Chapter 5 for case studies). This is a distinct contribution of our synthesis approach: it realistically deals with factual problems in software development such as requirements elaboration and elicitation.

The concepts behind mode-machines – as we use them in this thesis – were drawn originally from Hybrid Systems modeling [93, 139]. To the best of our knowledge, our approach is the first to (re)use

this concept to structure scenario-based specifications – taking a lateral-thinking approach compared to the traditional flow-charting approaches – and this distinguishes our work from others and opens new research avenues in this area.

In the rest of this chapter we provide a detailed survey of design-model synthesis approaches and we provide comparisons where necessary.

3.3 Behavior Models Synthesis – State-of-the-Art

In this section we discuss the related approaches of synthesis methods and techniques. Our target are those approaches that (semi-) automatically generate formal automata models out of intuitive specifications such as interactive specifications and declarative specifications (See Chapter 2). We exclude those approaches that assume formal models themselves as input to the synthesis process (c.f., [41]), because these input models are, by definition, amenable to analysis-techniques. So any further synthesis efforts will be a type of model-transformation. On the other hand, those models are too complex for manual specification, particularly at the early stage of development when detailed information is not available to specify these models. Additionally, the specification languages used are infamous of unintuitiveness and that is a main motivation in our research here.

We analyze the approaches from different aspects and we evaluate the related approaches with respect to each aspect by discussing their strengths and limitations. In Section 4, we discuss the gap in the surveyed approaches and how our solution attempts to fill this gap.

This survey is not intended to be exhaustive and is not result of a systematic literature review. Rather, it attempts to discuss the strengths and weaknesses of the mainstream approaches that have reported results and have had an impact in shaping this research area. Hence, the discussion in this chapter can well be used as a reference for further research that addresses other aspects that might not have been included in this thesis.

3.3.1 Input to the Synthesis Process

One of the major distinguishing factors in synthesis techniques is the descriptive form of input specifications. Most of the existing approaches base their techniques on huge assumptions about the input specifications and, in almost all cases, assume a subset of the features supported by those specifications methods. This is a natural consequence of the diverse varieties of scenario-based specifications [27, 28, 38, 72, 103, 123]. However, a technique should not be too constraining; otherwise, it will be impractical for non-trivial specifications. In the following subsections we detail

those assumptions and their limitations.

3.3.1.1 Formalisms

Most scenarios description methods are informal. However, in practice, the downside of informality is outweighed by the pictorial intuitiveness of scenario description form. Most approaches to model synthesis assume basic Message Sequence Charts (MSC) (c.f., [9, 18, 85, 91, 132, 133]) due to its extensive base of features and constructs. For similar reasons, we have given special attention to MSC scenarios. We detailed their features in Chapter 2 and we have provided analytic discussions on the extent to which each feature helps in model synthesis from scenarios. Algebraic semantics of MSC scenarios are provided in [97, 119].

UML Sequence Diagrams SD comes next in popularity within model synthesis approaches [37, 44, 127, 145, 146, 149]. SDs are informal and this led the relevant synthesis approaches to assume SDs with annotations of logical assertions, such as OCL [103], to allow some analysis of these SD scenarios.

Message Sequence Charts MSC are the most preferable scenario types adopted by synthesis approaches. There are few notational differences between MSCs and SDs, though the latter is too much oriented towards the OOD terminology [103, 116] (see Chapter 2 for detailed discussion). Although the telecoms standard, MSC-96 [69], has provided algebraic semantics to MSCs, for several reasons we do not see it changing the way the MSCs are actually used. First, MSCs are intended for system-level specifications and their users are not necessarily software-engineering experts, whereas SDs that are more likely to be used by computer science graduates who are (ideally) capable of using such algebraic semantics. Second, the unavailability of tooling support for semantic-based analysis of MSCs does not allow MSCs to become more popular than other forms of scenarios. This is unlike SDs that get the support of UML-based tools with testing and simulation capabilities (e.g., IBM Rational® tools).

We take a lean approach to introduce rigor in scenario-based specifications, for the purpose of synthesizing an automata model from these specifications. Our view is that scenarios should maintain their original spirit of simplicity and intuitiveness, which has boosted their popularity among developers and that they should not be stretched to lengthy and complex specification artefacts. We believe that the formal component in specifications should be a different, complementary type of artefact which includes sound formalism. Such type of artefact should augment scenarios in order to constitute an overall specifications that cover the system's behavior. To this extent, we use modes (see Chapter 4) to introduce that formal component in scenario-based specifications. This leads to the idea

of combining complementary types of models in the same specifications, which we discuss in the following section.

There is a wealth of other theoretical work on the realization of MSCs and hMSCs (c.f., [10, 16, 49, 50, 51, 52, 64, 92, 102]). Although these approaches have achieved fundamental results in this area, they are rather theoretical and they are not directly related to the work involved in this thesis. We will however refer to these approaches, where relevant, in later sections.

3.3.1.2 Hybrid Specifications

The tedious task of specifying software systems has led researchers to conceive several *perspectives* (a.k.a. *views*) of the system behavior, and in turn this resulted in hybrid specification artefacts. This is a classical idea [84] and has led to several research directions. In model-synthesis research, this hybrid approach to specification is actually both a challenge and an opportunity. For example, inconsistencies between the different model types could easily arise as these artefacts are manually specified. On the other hand, hybrid specifications are complementary and allow specifying more detail about system behavior and this offers a potential to help reduce synthesis processing.

Sun and Dong [129] combine LSC scenarios and Z artefacts [38, 148] and use them collectively as input to their synthesis approach. Z is a state-based formal specification language based on set theory and first-order logic. Z is used in [129] to model state-based behavior of individual components. The main motivation is that Z specifications complement the lack of expressiveness of LSC to capture local actions (i.e., self actions on the same object) [38, 129].

Uchitel et al. [132, 133, 134, 136] complement basic MSCs with safety properties expressed as Fluent Linear Temporal Logic [89]. Their motivation is that scenarios are existential behaviors that provide acceptable behaviors of the system, while properties provide universal statements that must be satisfied by all elicited behavior. So, in [133], properties complement scenarios by *pruning* the space of acceptable behavior.

Other research approaches [83, 146] implicitly use OCL-like format (OCL is Object Constraint Language, [103]) as annotations to scenarios. The basic idea of annotating a scenario with these assertions is to enrich the scenario with state-based information and also to provide a means of checking the consistency of the scenario's flow. These annotations assume some domain theory of the system, i.e., a set of domain variables and their associated semantics and constraining predicates or assertions of the values of these variables. Annotating scenarios with domain assertions helps to generate state-based automata models, instead of labeled transition systems (LTS), [78], that inherently have a higher number of states than their state-based counterparts.

The synthesis approach of Whittle and his colleagues, [145, 146], assumes non state-based models as complementary specifications to scenarios, which is a special case among the approaches we survey. They use the UML Class model to help reason about and generate a hierarchical state machine model in a form of statechart [59, 61].

From the discussion above, we notice the following:

- A common denominator of all approaches is that they use state-based specifications to express component-wise state information which have little or no support in known scenario description methods (e.g., MSC State Labels [72] are too abstract to help in model synthesis). However, no approach has attempted to augment scenarios for the purpose of overcoming their inherent partiality problem.
- Some approaches (c.f., [129] and [133]) use explicit state models. However, none of these state models has direct relationships with the scenarios themselves. For example, Z specifications used in [129] are associated with individual components, but are not related to scenarios. Also, the temporal Logic properties in [133] are system-wide constraints, requiring scenarios to satisfy them, but do not have, for example, a one-to-one relationship with scenarios.

What we seek here is a powerful combination of state-based and scenario-based specifications types such that:

- Firstly, each specifications type should be employed within its expressiveness limit, without overuse that could lead to adverse results. For example, very long scenarios or over-conditioned scenarios (i.e., scenarios with too much nesting of constructs such as **loop**, **alt**, **par** (See Chapter 2 for detail) are no longer readable, tractable or maintainable by engineers. Also the state-based part of the specifications should be abstract-able to suit the early stages of development, when little information is known about the system.
- Secondly, a *modeling relationship* between the two types of specifications should be exploited so that each type becomes integral to the other. A good step in that direction was the insight offered by Uchitel et al. [133] that assumes properties are universal behaviors that prune the acceptable behavior space provided in scenarios. However, in our opinion, this does not fully reflect the potential of mixed-type specifications to be a powerful behavior requirements technique, which, in particular, supports synthesis of detailed automata models.

In our work, we propose a synergistic relationship between scenarios and an abstract form of automata models known as *modes*. We use *modes* to provide contextual scope to scenarios in a one-to-one (and

possibly one-to-many) mode-to-scenario(s) relation (see Chapter 4). We exploit this relationship and use it to propose a novel framework for structuring the scenarios specification (see Chapter 5) for the purpose of requirements elicitation, on one hand, and hand to facilitate synthesis of refined automata models on the other (see Chapter 6) from the integrated modes and scenario specifications types.

3.3.2 Generated Models

By output models we mean the detailed models of behavior that are generated from the model during the synthesis process. These models, sometimes referred to as *design models* [133], are effectively derived from the requirements specifications models that we discussed in the previous section. The derivation procedure and algorithms are determined by the specific synthesis process used. To avoid ambiguity, we will use the terms *design models*, to refer to these output models generated by the synthesis process, and *requirements models*, to refer to the input specifications models.

The design models involve several decisions that could well be platform-dependent; this is highly determined by the synthesis process and the thesis behind its approach. The intuitiveness of these decisions, the way they are introduced in the model, and how they contribute to further elaboration of the models, are among the distinguishing features of the approaches we detail below.

3.3.2.1 Formalisms

Synthesis approaches vary in their selection of the formalism of the generated design models, however, almost all those formalisms are automata-based. Some approaches assume event-based automata formalisms and most of these formalisms are based on the Labeled Transition Systems LTS [78] and its variants: PLTS [136] and MTS [83, 133]. Other event-based formalisms are also used (e.g., Bontemps et al [20, 21] use I/O Automata).

Fewer approaches [39, 60, 85, 146] use state-based formalisms where states (and, in some cases, transitions) are labeled by propositions and predicates. A popular state-based formalism is the statecharts [59, 61] which has a rich set of syntactic features, in addition to its intuitive modeling principles. However, these features come at a price: it is easy to write down ambiguous statechart specifications.

Damas et al., [37], uses a mix of event-driven and state-based models where the states are 'decorated' with assertions on the state-variables.

Obviously, the synthesized model's formalism is dependent upon that of input specifications models. Approaches that embed OCL-like assertions in scenarios, typically synthesize state-based

models. On the other hand, approaches that assume MSCs with abstract message labels, typically synthesize Labeled Transition Systems. So, a careful selection of the input specifications (particularly if using mixed types) is crucial to determine the procedure itself, as well as the synthesized output models.

3.3.2.2 Synthesizing Global Automaton

The interactions in a scenario take place between a set of components (or objects). These components typically belong to the system under development and possibly to external components from the environment in which the system is designed to run (e.g., the system user). Most synthesis approaches build an automaton for each component, such that this automaton exhibits all behaviors of its associated component that appear in all scenarios where this component is involved. A few approaches, e.g. Damas et al. [37] and Harel et. al [60] build a global state machine of the entire system at early an stage of the synthesis procedure, which is likely to cause a state-space explosion. Moreover, constructing global automaton is not feasible in the case of a distributed component system, where each component is a subsystem on its own, with a separate state-space.

It's worth mentioning here that if the entire system participates in scenarios as a single black-box component, then the generated automaton of this component will reflect the behavior of the whole system without details of its internal state (e.g., states that result from local or internal actions). These issues are related to the modeling concepts of system observability and controllability, and are highly dependent upon the system specifiers to decide the extent of observability of the system states.

3.3.2.3 Abstraction

For non-trivial systems the synthesized models are likely to be too large to be readable for manual inspection or presentation. So, the generated model has to be *abstract-able*, but not abstract: the synthesis process is effectively a refinement process of the input specifications with more technical design decisions resulting in a design model. It worth mentioning that we do not mean that the generated model will necessarily be platform-independent.

The work by Whittle and his colleagues [146, 145] is distinguished by their attempt to address this problem and generate statecharts models that promote hierarchic state machines. However, auto-generating hierarchic state machine as part of the synthesis process is tricky and could easily lead to ambiguous models (given the semantics issues with statechart and its variants [143]). For example, the heuristics in [146] may generate readable hierarchic statechart, but it can easily generate unambiguous ones, as it heuristically exploits generalization relations among variables in the state

vectors' annotations on the used UML SDs in [103]. Moreover, the work of Whittle and Jayaraman [145] depend on the Interactive Overview Diagrams (IOD) [103], that recently introduced in UML 2.0. Effectively, IODs are flowchart-like diagrams that prescribe an ordering of UML SDs with some new features, such as *preemption*. Whittle and Jayaraman exploited the structure IODs to generate hierarchical state-machines, instead of flat machines as done in previous related work [146]. For now, we can say that the flowchart-based structuring concepts in IODs are inherited by the state machines generated from IODs as reported [145]. These flowchart-based concepts do not promote the basic concept of abstraction [104]. Moreover, that statecharts concepts of state hierarchy are questionable [105].

We detail in Chapter 5 a discussion on the effectiveness of IODs as a technique for *scenarios structuring*. Moreover, in Chapter 6 we demonstrate the use of these structured scenarios as input to the synthesis process.

The approaches of Uchitel et al., [133], and Krka et al., [83], generate Modal¹ Transition Systems (MTS) [87] as output models. Simply speaking, an MTS is a labeled Transition System (LTS), [78], with the ability to distinguish “May-be” and “Must-be” transitions. MTS models assume the abstraction lies in the “May-be” state transitions, in the sense that they hide the (as yet unknown) decision of keeping this transition (i.e., label it as “Must-be”) or removing it from the model. If we look at the basic concept of abstraction we see this concept of “may-be”/“must-be” is not aligned with it, though it is useful.

Sun and Dong [129] use predicate abstraction [15] to construct abstract machines for Z-packages specifications (which is an input to their synthesis process), and use those machines in the refinement stage of their process. The purpose in [129] was not to introduce abstraction in the output model, but rather to refine (and constrain) the behavior in those other machines generated for each object appearing in the input scenarios.

We take a more proactive approach to introduce abstraction in the synthesized output model. We use abstract machines called *modes* as part of the input specifications in the first place. Mode Machines (Chapter 4) are state-based abstract specifications, that exploit the system state variables to model the state-space at its outset and from different aspects. The high abstraction level, the aspect modeling and the disciplined specification of these machines (see Chapter 4) overcome the modeling complexities of flat state machines or LTS-based automata [78, 87, 136].

We use these machines, which are codified as predicates over state variables, as an integral part of the input specifications, where they are combined with scenarios in a synergistic relationship (see

¹Note that the concept of *modality* in MTS that originates in modal logic [30] is completely different from the concept of *modes* that originates in Hybrid Systems that we employ to structure scenario specifications in our work.

Chapter 5). Consequently, our synthesized models inherit a detailed version of those machines resulting in hierarchic abstract state transition system that are straightforward enough to avoid the semantic problems associated with statecharts [59, 61, 143].

3.3.2.4 Refinement

Synthesized models should also be amenable to refinement leading to an implementation of the system-to-be – otherwise they will be throw-away models. Synthesis approaches need to provide a proper refinement method to direct the synthesized model from its abstract form to some implementation. The synthesized model should have the basic ingredients to be amenable to refinement to detailed models, possibly up to the implementation form such as source code. There are a plethora of refinement notions and calculi associated with corresponding automata formalisms.

The approaches [83, 133] that synthesize MTS models assume a post-synthesis step to refine (or, to put it better, *resolve*) the “May-be” behaviors in the MTS to either “Must-be” behavior or to remove it completely from the model. After some automated analyses of the synthesized models, a decision is taken to keep or remove the behavior according to the results of the analysis.

Most of the generated models are, more or less, standard automata and can be systematically refined to an implementation form using industry-applied code-generation techniques [122]. However the synthesis processes do (including ours, in Chapter 6) not cover the implementation step because the scope of the synthesis process is to generate the models and further refinement is likely to involve platform dependent decisions, which are out of scope of the research work under discussion.

3.3.3 The Methodical Aspects of Synthesis Process

3.3.3.1 Algorithms

The synthesis algorithm is central to the synthesis approach. Because each approach makes assumptions on which the method is built, it is necessary that an approach provides detail of its procedure or algorithm to crunch the input specifications and generate the output state machine. There are several distinguishing aspects we can use to classify the diversity of these algorithms. In this section we examine two aspects: the foundation of the algorithms, and their monotonic complexities.

On the foundations: some approaches [85, 127, 133, 134, 137, 145, 146] use algorithms that demonstrate and implement theorems and assumptions developed by the authors themselves. Sun and Dong [129] apply algorithms based on the *observability* and *controllability* concepts – drawn from the

Supervisory Control of event-based systems [29] – in order to prune the intermediate state machine they generate in their synthesis process [129]. Bontemps [19] uses a game-theoretic solution in his synthesis approach to generate the output as Linear Temporal Logic (LTL) [114]. The same author [20] uses existing theoretical foundations of Buchi Automata.

Other approaches [18, 35, 37, 94] apply *learning algorithms* [12, 40] to infer the output synthesized models. One of the earliest attempts to exploit learning-based techniques for interactively synthesizing models from a given set of examples was the work of Makinen and Systa [94]. They proposed the Minimal Adequate Synthesizer (MAS) algorithm and tool to interactively assist designers to build a statechart model for each process. The MAS algorithm is defined in terms of grammatical inference and it uses Angluin’s algorithm [12] to infer a statechart with the help of *membership* and *equivalence* queries [94] to be answered by the designer. The major drawback of MAS is that composing the resulting statecharts in parallel yields a system that may exhibit undesired behavior or deadlocks.

Damas et al. [37] use an interactive procedure of classifying positive scenarios (ie. desired scenarios) and negative scenarios (i.e., scenarios that should not happen) and derive an LTS for each process. They use *passive* learning algorithms (c.f., [40]) to infer a global intermediate model, which is subsequently transformed into a distributed system. The *passive* learning algorithm employed by [37], however, does not support incremental model generation in terms of extensions or amendments of the given requirements specifications.

Another approach based on passive learning algorithms is that of [35], where it also uses grammatical inference to derive a formal model of a process from a given stream of system events; however, this approach works only with positive event streams (which correspond to positive scenarios).

Bolling et al. [18] employ Angluin’s algorithm [12] in their approach. The basic algorithm in [12] is used to infer deterministic finite automata from regular languages [8, 67]. Bolling et al. extend this algorithm to synthesize Communicating finite-state machines CFM from MSCs [72]. They consider an MSC as a partial order on events from the system’s alphabet.

Other approaches rely on *inductive reasoning*. While *deductive reasoning* arrives at a specific conclusion based on a given set of generalizations, *inductive reasoning* is essentially at the opposite side where it attempts to create general principles by starting with many specific instances.

Process phases: We may also differentiate between the algorithms associated with different steps or stages of the synthesis process. There are however tricky steps, such as merging the (partial) state machines that are independently generated from separate scenarios of the same component into

a single (integrated) state machine representing the overall behavior of the system. The basic idea of merging those machines is to efficiently identify identical states. Another challenge is the emergence of non-determinism.

Whittle and Schumann [146] identify identical states in different FSMs by matching their state vectors (i.e., the valuation of the state variables' vectors in each state). They join identical states by, first, connecting them with empty ϵ -transitions and join them by applying a standard algorithm from [5] to remove these ϵ -transitions and simultaneously merge similar nodes. This is the most basic approach to FSMs merging.

Uchitel et al. [133] use a more sophisticated algorithm to merge their MTS [87] based models. They search for the *least common refinement* [133] model which is, roughly, a single MTS machine model that exhibits all the behaviors of the merged MTSs without contradictions (and sometimes it helps to self-refine the MTSs when a “May-be” transition in one MTS is “confirmed” by an identical “Must-be” transition in another MTS).

In our approach, since we have several aspects to consider in our model merging process, we merge our mode-based automata models in two steps. First step: we refine each mode-machine along with its “embedded” FSMs generated from scenarios associated with this mode machine. For each mode-machine, this refinement involves merging each subset of FSMs into a single FSM (and the mode-machine itself resolves in the resulting FSM). Second step: we apply a standard process, similar to that of [146], to merge the resulting FSMs (output from the first step) into a single machine.

Complexity: since the synthesis process has an automated component, complexity of the algorithms involved becomes an important factor. Bontemps [19, 23] explored the classic *Realizability Problem* [115, 1, 57] in the context of scenario-based specifications and proposed a very high complexity algorithm (triply exponential). In an improved version of this work Bontemps et al. [22] abandoned the completeness and proposed lightweight algorithms that still retain soundness. A comparison of the various types of algorithms employed in different stages of the synthesis process, and their complexity is provided Bontemps et al., [21].

Our synthesis algorithm, presented in Chapter 6, has polynomial complexity. However, since our synthesis approach belongs to the *compilation* category of synthesis approaches, it is unfair to compare it to the exponential-complexity of other approaches in other categories. See Section 3.2.2 for description of different categories of approaches. Compared to other approaches in the same *compilation* category: Uchitel et al. [133] proposes an exponential algorithm to synthesis MTSs from properties expressed in Fluent LTL, and another second-order algorithm to synthesize MTSs from

scenarios.

3.3.4 State Space coverage

We propose that the major feature distinguishing our synthesis approach is state-space coverage that helps to minimize the partiality in requirements specifications. The *behavior space* of the system is well defined and bounded by its (automata-theoretic) state-space, which in turn is defined by the system's state variables (See Chapter 4 for more detail). Because scenarios, by definition, are examples of system behaviors, they are partial specifications of the system, completeness of scenario-based specifications is hard to ensure if we judge it intuitively by writing more scenarios. We take this problem of incomplete specifications seriously in our approach and use *modes* (Chapter 4) to augment scenarios in order to help uncover hidden gaps in the state-space (see Chapter 5 for detail of this method).

To our best of knowledge, no other approach has attempted to address state-space coverage by structuring partial specifications with respect to the state-space, as we do in this thesis (see Chapter 5.)

3.4 Discussion and Summary

In this chapter we have discussed in this chapter a wide range of approaches to automata-based model synthesis. We discussed the contributions of these proposals in detail illustrating the trends and directions researched for improving design models synthesis.

We surveyed the types and forms of the input specifications (i.e., requirements) from which the design models are synthesized. We also looked at the various formalisms assumed by these approaches for the synthesized output models, and finally we looked at the methodical aspects of these approaches, such as the proposed algorithms, their foundations, complexity, etc.

Clearly, almost all approaches focus on scenarios, as we do here, but few of these approaches attempted to address the problems associated with these specifications, such as partiality and composition. Some approaches have used special forms of scenarios such as LSCs [38] and Triggered MSCs [123]. Other approaches considered composite forms of scenarios such as High-level MSC [72]. Neither of these scenario forms helps to ameliorate the partiality of scenarios or to provide appropriate structure in scenarios. (High-level MSCs compose scenarios in a flowchart-like form that is not suitable as an architecture artefact [104]). In our synthesis approach we contribute a novel technique to structure scenarios specifications using *modes* [7, 63] (see Chapters 4 and 5). *Modes* provide a

multi-view, automata-based structure of the system’s state-space. Scenarios can then be specified within these modes that serve as *contexts* for scenarios and help guide their specification. and so maximize the coverage of behaviors.

The synthesized models also varied in the formalism and abstraction. An ongoing trend is to use the special automata formalisms of Modal Transition Systems (MTS) [87] – an extension of the old Labeled Transition Systems (LTS) [78]. MTS can be considered as a flavor of 3-valued logic. It assumes the possibility of “May-be” transitions that can be elaborated later to a “Must-be” or completely removed from the model. Such a *decision delaying* feature is helpful; however, it is questionable how this formalism will capture abstractions, such as predicate abstractions and how it will provide a refinement mechanism for it. Our approach uses *modes* to create abstract automata models (Mode Machines, Chapter 4)

It is noticeable that very few approaches synthesize state-based models, and most of the proposed approaches synthesize event-based automata models. Using event vs. state-based models largely depends on the target application domain. For our approach here we synthesize a standard state-based automata models like those of classic *Kripke structures* [82]. This is mainly because we target control and cyberphysical application domains where it is quite common to use state variables and state-based automata as behavior models.

In the next two chapters, we will present our approach to structure the input specifications by using *modes* and apply a novel algorithm to synthesize, via a step-wise refinement, a concrete finite state machine out of these structured specifications.

Chapter 4

Behavior Modality

4.1 The origins of modes in Hybrid Systems

Wherever continuous and discrete behaviors meet, hybrid systems arise. Generally speaking, a Hybrid System [93, 139] is a mixture of continuous and discrete behaviors. These continuous and discrete behaviors not only coexist, but also interact together when the system state changes in response to the execution of both the discrete and continuous behaviors.

The roots of hybridism in the behavior of a single system stem from pure physical systems. For example, a ball bouncing on a flat horizontal surface will have different dynamics (i.e., behaviors) while it is falling, rising or striking the surface. These different behaviors are separately modeled as different continuous behavior-models and the transitions between them are modeled as “state jumps”. We refer the reader to [93, 139] for a deeper look at hybrid systems theory and applications.

Computer scientists, in particular, are interested in those hybrid systems where the discrete parts of the system are digital. This is especially profound in many technological systems, in which decision-making and embedded control logic are combined with continuous physical processes [88].

In this section we discuss a central modeling element of hybrid systems: *mode*. Intuitively, a mode is an abstract state that identifies a unique behavior; a hybrid system is said to be in mode A, if it is currently executing the behavior associated with mode A. The system has a number of modes – as many as the number of unique behaviors it exhibits. While in a particular mode, say mode A, the hybrid system may take any state of the state set that characterizes the particular behavior associated with mode A.

4.1.1 Behavior Hybridism and the Role of Modes in Hybrid Systems

Identifying the role of 'modes' is simply a case of identifying the different behaviors of the system. In other words, a hybrid system that exhibits a number, n , of unique behaviors is said to have n modes of operation; and a hybrid system is said to be in mode M , if it currently executing the behaviors associated with M . The real challenge, however, is to identify the different behaviors that make up the overall system operation, and this is usually a modeling problem. We take a look below at different situations of this hybridism in system behavior.

Hybridism may arise due to discontinuities in the system's input/output behavior, sometimes depicted as "jumps" in the trajectory graph of this function. For example, the behavior of a bouncing ball, plotted in Figure 4.1, has discontinuities in its execution trajectory, which causes non-linear "jumps" in the state succession path. This situation of hybridism is well-known in physical systems with nonlinear dynamics. Traditional mathematics and calculi are unable to model such abrupt changes and nonlinearity. So, the overall behavior has to be broken down into a set of partial behaviors that are "regular" enough to be described concisely by a mathematical model. Such a mathematical model is commonly known as the system's state equation [139]. The set of partial behaviors is identified by a corresponding set of modes, and a mode-transition relationship is specified, beside the continuous behaviors, to determine the switching logic between the system modes. This transition relationship is typically a simple sequential relationship in case of physical processes. The modes and their transition relationship represent the discrete side of the hybrid system.

Consider the physical dynamics of a bouncing ball where its state variables (position, speed, acceleration and direction) suddenly change from one mode to another while the ball is bouncing. The behavior is modeled by the following piecewise continuous equations:

$$v(t) = \begin{cases} -\dot{p}(t) \text{ m/s} & \text{while in rising mode} \\ p(t) \text{ m/s} & \text{while in falling mode} \end{cases} \quad (4.1)$$

$$\dot{v}(t) = \begin{cases} -9.81 \text{ m/s}^2 & \text{while in rising mode} \\ 9.81 \text{ m/s}^2 & \text{while in falling mode} \end{cases} \quad (4.2)$$

where p is the ball's position, v its vertical velocity and 9.81 m/s^2 is the constant gravity acceleration. This function is known as the state-space Equation of the system and is used in traditional systems theory as mathematical blueprint or model of the system behavior.

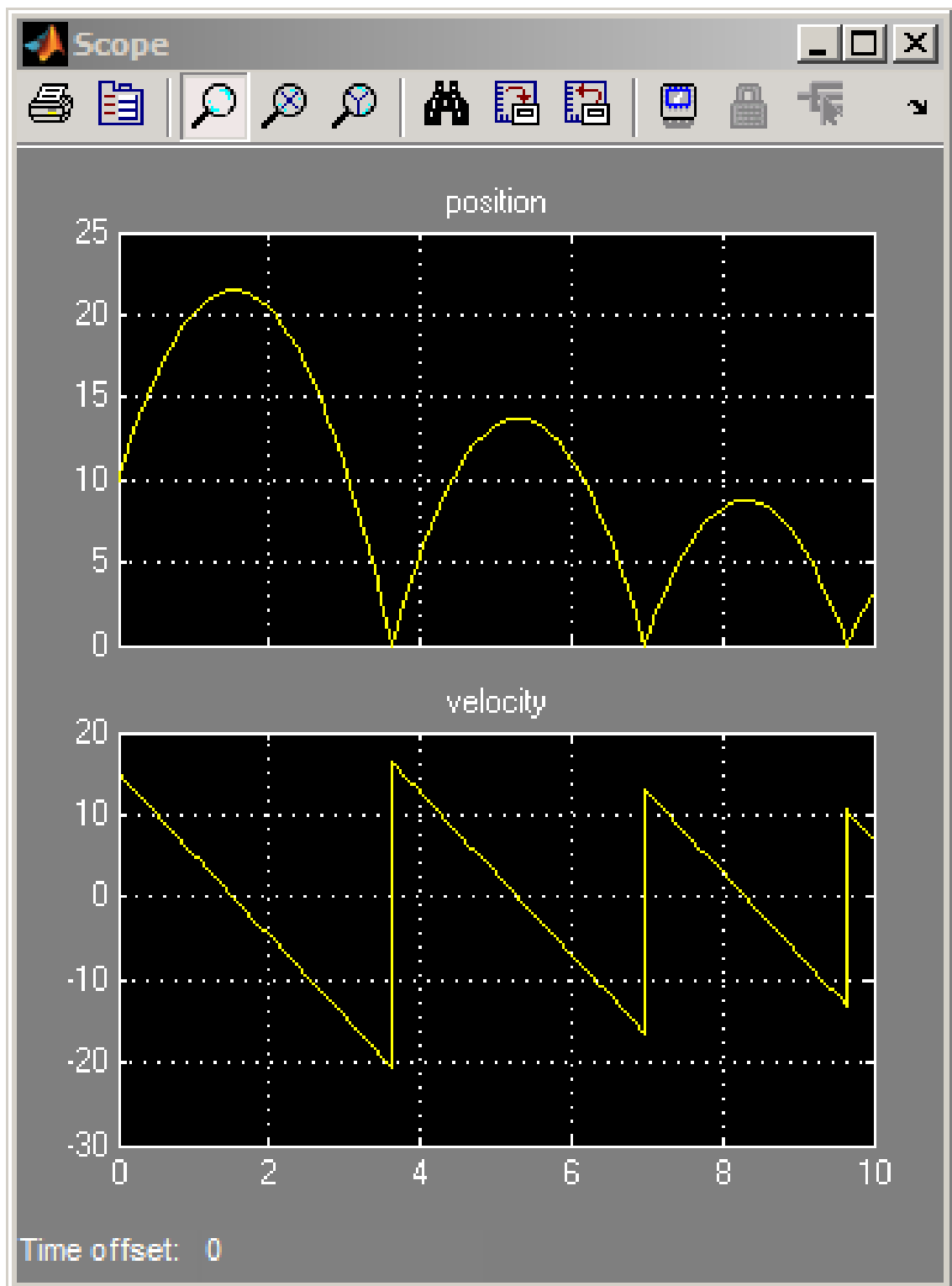


Figure 4.1: Hybrid System example: bouncing ball behavior trajectory

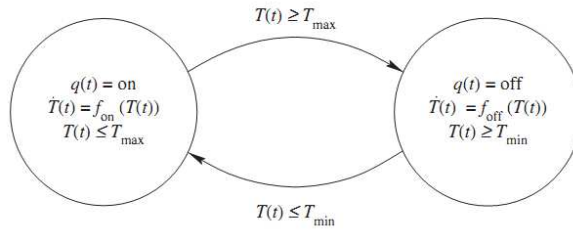


Figure 4.2: Technological System example: Temperature Controller

While the ball is in falling mode it changes its state uniformly, according to the gravity’s acceleration, until it strikes with the surface where it changes its mode from falling to stop and then to rising. At the point of impact, the ball’s state variables change completely causing the ball to enter a new mode: the speed flips instantly from the highest value it reached to zero and then instantly back to high; the direction flips upward; and acceleration flips to deceleration (because of gravity). While in any of its modes, the ball’s state changes regularly according to the state-equations of this mode.

4.1.2 Moving From Hybrid Physical Processes to Technological Systems

Technological Systems, where decision-making and embedded logic are combined with the continuous behavior of physical processes, are other examples of hybrid systems that are of interest in the computing domain. These technological (and hybrid) systems are pervasive in our daily life. They range in complexity from simple thermostats to embedded car-engine control systems. The behavioral hybridism in these systems is artificially introduced via embedded logic in order to control a physical process, for example, to maintain a desired operating point. This is unlike the hybridism in the behavior of the bouncing ball where the behavior naturally emerges and the ball autonomously switches its behavior, from a mode to another, at the point of impact. The common denominator of both types of hybrid system is that switching between the different modes forms a machine of modes that serves to “overarch” the system behavior and its state-space. Consider the example of a simple thermostat in Figure 4.2.

The temperature-controller model in Figure 4.2 comprises two modes; the heating mode and the cooling mode. The heating mode is characterized by the predicate equation $q(t) = ON$, and the cooling mode is characterized by the predicate equation $q(t) = OFF$. These modes identify

two different behaviors characterized by the state-equations

$$\dot{T} = \begin{cases} f_{ON}(T(t)) & , \text{when } T(t) \geq T_{min} \\ f_{OFF}(T(t)) & , \text{when } T(t) \geq T_{max} \end{cases} \text{ respectively.}$$

These two state-equations describe the continuous behavior of the thermostat. On the other hand, the two modes’ identifiers (heating and

cooling) and the logic of switching between them describe the discrete behavior of the thermostat. Switching from cooling to heating modes takes place at time t_{ON} , such that $T(t_{ON}^+) = T_{min}$. Similarly, switching from heating to cooling is driven by the event $T(t_{OFF}^+) = T_{max}$. Figure 4.2 shows the discrete side of the thermostat system model.

Modes are pervasively used in hybrid systems' models as first-class elements to manage the complexity resulting from the diversity of behaviors, and to facilitate the use of standard techniques to analyze and control of these behaviors. We presented two examples of hybrid systems where modes are used to model the discrete side of the behavior for different purposes.

In real world systems, particularly in technological systems, there is typically more than one aspect of hybridism in their behavior (e.g., the behavior of physical versus computing processes); the individual behaviors are interdependent and sometimes overlapping under certain conditions. Such complexity is no longer manageable by a simple mapping-function from a set of distinct behaviors to a set of modes (as demonstrated in the bouncing ball example 4.1). These cases arise in highly discrete systems, such computer software, where the behavior is determined by the set of user requirements that are typically non-homogeneous.

In the rest of this discussion we look at these cases and incrementally elaborate on the Modal Behavior modeling of hybrid systems into multiple-degrees modeling of Modal Behavior.

4.1.3 N-Degrees of Modality in Behavior: Engineering Software with Modes

In most physical processes with hybrid behavior, identifying the system modes is fairly straightforward, given the continuous behaviors' specifications. Simply, each distinct behavior is characterized by a mode (see the bouncing ball example 4.1) and the system switches between those behaviors at runtime in response to input events (e.g, when the bouncing ball hits the floor). No challenging issues have been reported in identifying the system modes for singly-dimensioned Modal Behavior in traditional hybrid systems [93, 139].

In technological systems, however, system requirements go beyond a simple switching between exclusive set of modes. For example, most control systems consider a set of failure modes that coexist along with the core modes-set that identify the hybrid behaviors of the physical process being controlled. In software engineering community, a famous case study of this system type is the Steam Boiler Controller (SBC) [105]. In the SBC, the computer controller must maintain the water-level within certain limits using a set of pumps, and must also consider failure modes such as normal, degraded, emergency, etc. The core behaviors of the SBC are related to turning ON and OFF a group

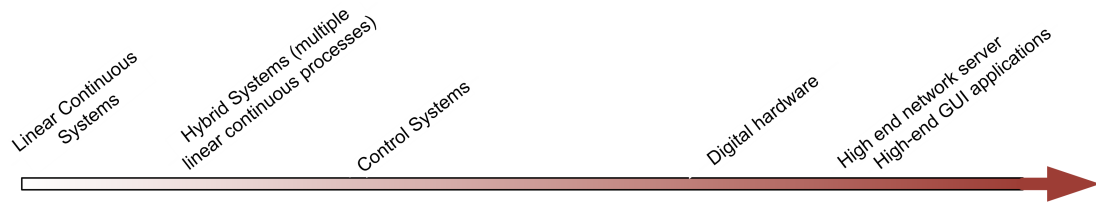


Figure 4.3: Behavior Irregularity Spectrum

of pumps, based on some hysteresis, to control the process of water level change. Along with the water-level control behaviors, the SBC has other behaviors to handle the failure modes. Such a set of failure modes in control systems is not additive (or exclusive) to the core system behavior modes but rather it co-exists and overlaps with the core behavior mode-set, forming another dimension of modality in the overall system behavior.

4.1.3.1 The behavior irregularity spectrum

Figure 4.3¹ depicts the spectrum of irregularity in engineering systems, with some common example systems annotated along the line.

Systems of linear and hybrid physical processes are at the lower end of the spectrum. Next there is a range of control-based systems, involving software and hardware. Even though they include software components, the software in these control-systems is typically implementing control-algorithms that have a limited effect on the overall system mode (e.g., an algorithm starts and finishes execution within the same system mode). Non-trivial systems implemented in digital hardware have more irregularity than the previous systems. Indeed, digital hardware has a huge number of discrete states, compared to control systems and nonlinear physical processes. When we move further towards systems with more irregularity and hybridism in behavior, software systems come to the picture.

4.1.3.2 The “surprises” of software irregularity

The basic challenge in modeling software systems is its unexpected behaviors, sometimes known as behavior surprises [106]. The extreme irregularity in (the requirements-driven) behavior impedes designers from comprehending the overall system operation. Software designers are still unable to sketch a (wishfully mathematical [108]) model as concise and precise as classical mathematics served

¹The classification in Figure 4.3 is relative and neither accurate nor comprehensive. Our goal is it to put software systems and traditional engineering systems in one picture to illustrate the differentiating issue with modeling software systems. This helps motivate our use of modes to overcome this irregularity in software, as we will see shortly.

for traditional engineering systems, even though it is claimed that discrete mathematics serve the function that continuous mathematics did for traditional engineering.

One of the main reasons software systems are highly irregular is the unpredictable environments in which these systems live. For example, for nontrivial GUI applications, designers can not predict (all possible) sequences of manipulating the GUI controls; a communications server does not know what message sequence could arrive during any period of time; etc. The design of those systems usually captures the typical cases inferred from requirements (plus some ad-hoc selected boundary conditions), leaving the rest of “case-space” as unexpected surprises that hopefully won’t happen. Some designs handle the unexpected events by ignoring them [110]. This has resulted in a software engineering dispute which has escalated to the extent that researchers believed software development is a radical design task [74] to which normal engineering [110] concepts are not applicable.

The concept of modality is sufficiently extensible to manage such irregularity in software, in much the same way as it did for hybrid systems. Software systems are necessarily multi-aspect, and thus modeling their behaviors requires N-degrees of modality.

4.1.4 Section Summary

We presented the historical background of the origins of modes in hybrid systems, and we drew together these classical hybrid systems and modern software systems in one picture. We tried in this section to introduce modes. In general, we indicated how modes can manage the complexity of software systems, from the traditional engineering viewpoint. Recent calls for specializations [73] in Software Engineering might help to extract some universal properties of those specialized domains. In section 4.2 we try to link this characteristic of software systems and use modes to address it.

4.2 Achieving N-Degrees of Modality via N-Classes of Modes

So far, we outlined the role of modes in managing behavior irregularity in physical systems, as well as their potential to serve a similar role in software systems. As software systems are multi-aspect systems, the one-dimensional view of modes in hybrid systems needs to be extended to a multidimensional one in order to cope with several aspects typically arise in software systems. In this section we discuss the mechanisms to achieve N-Degrees of modality, building on existing research work.

4.2.1 What makes a good abstraction

Abstraction is the ultimate solution to manage this (irregularity-driven) complexity in software systems. The key point, however, is to find useful and “faithful” abstractions. A System Model is an abstraction of the real system, revealing some properties of interest and hiding other details. Usually, the revealed properties are those that are implementation-independent, while the reverse is true for the hidden properties. A useful abstraction, in our opinion, is the one that is simple enough to be consumable by humans and formal enough to be analyzable by a computer. The classical example of such an abstraction in computer science is the transition diagram [67]. A faithful abstraction does not show properties which the system does not possess in any possible implementation.

4.2.2 Mode-Classes: Reinventing “Modes” in Software Engineering

David L. Parnas brought the idea of multi-dimensional modality to the paradigm of software engineering [7, 36, 107, 109, 113]. Even though the idea was developed to model behavior complexity in control-related applications, it is equally applicable to other problems that are not necessarily control-oriented [14, 117]. This work started at the Naval Research Lab where Parnas headed the Software Cost Reduction (SCR) project to redesign a major part of the Flight Management Software of the A-7 aircraft, in an attempt to prove the feasibility of his formal approach to software specifications [7]. The basic concept of modes was already used by the A-7 aircraft engineering team, in the same sense of hybrid systems, but these modes specifications were singly-dimensioned and, so, too complex to comprehend [7]. Parnas attempted to restructure those modes, and their associated behaviors, to help describe the functionality of the flight management software using his functional-specifications method [36, 63]. Parnas clustered these modes into *classes of modes*. Each mode-class contains a set of modes that are exclusive to it, i.e., the system could exist in one and only one of the modes belonging to the same mode-class. However, modes belonging to different mode-classes are not necessarily exclusive.

In this thesis, we build on the basic idea of mode-classes, and we conjecture that this arrangement of modes has good potential to enable multi-dimensional behavior specifications, and is potentially applicable to most software systems and applications such as user interface, distributed systems, etc. The approach, as we envision it and apply it here, is independent of the specifications *form* (be it mathematical functions or informal scenarios). So it potentially could allow integration between different paradigms of software specifications. Our focus in this thesis is to apply this multi-modal approach to introduce structure into scenario-based specifications, with the goal of supporting the synthesis of automata-based behavior models of software applications. We consider this as a first step

for future exploitation of multi-modality in software engineering, opening another avenue of investigations beyond the work in this thesis.

4.3 Automata-Theoretic Formulation of Modes and Mode-Classes

So far in this chapter we attempted to, informally, flesh out the concepts of behavior modality and we have given a historical background supported by examples of real systems. In this section we provide theoretical foundations of these concepts and derive sound properties of model-refinement that are useful to the behavior synthesis technique we introduce in Chapter 6. Even though the concept of Modal Behavior (in its raw form) originates in hybrid systems, the mathematical formulation and the derived properties we provide in this section, are not replacing the theoretical foundations of hybrid systems (c.f., [93, 139]), but rather are a “port” of it to the domain of highly-discrete systems such as software applications.

In this section, first, we make an important distinction between the *state* and *mode* concepts – something rarely made clear in the computing literature. Also, we explain the concept of mode-class, in the light of the available literature [7, 36, 63, 109], and we show how the role played by *mode* has already been played by state but at a more concrete level. We also indicate the benefit of capturing several views of the system via *multiple classes of modes*.

Second, we put our formulation of mode-class into automata-theoretic form by defining a *Mode-Machine*. This helps to put modes specifications and the other specifications, such as scenario specifications (see Chapter 2), into one homogeneous form (i.e., automata), that allows integration and cross-refinement of both types of specifications into a single elaborated behavior model of the system which is amenable for further analyses. Finally, we derive important properties of modes, such as *modes refinement*.

In Chapter 2 we briefly outlined the Engineering Safety Actuation System (ESFAS) [2] and used it as a running example to illustrate scenario-based specifications. We will continue to use this system here as a running example to explain the concepts when necessary.

4.3.1 Mode vs State

We begin by recalling the familiar concept of system state, or simply state. Let V be a set of variables and D be a set of values. A state is a function $t : V \rightarrow D$, and a state-space $T = \langle V, D \rangle$ is the

set of observable states. The space T is defined as $v_1 : D_1, \dots, v_n : D_n, \forall v_{k \leq n} \in V$, and $\forall D_{k \leq n} \subseteq D$ where D is referred to as the domain of interpretation.

Assuming an arbitrary ordering of variables, a state, $t \in T$, can be given as a tuple; $t = \langle t(v_1), \dots, t(v_i), \dots, t(v_n) \rangle$, where $t(v_i)$ is the valuation of the variable v_k in the state t , and n the number of variables in T .

We assume the state transition system as the concrete level of space representation, relative to which we describe more abstract representations using modes.

Definition 1. (Mode M). Let $T = \langle V, D \rangle$ be a space and let Q be a predicate over V . A *mode* M is a subset of states $M \subset T$ such that $\exists t \in T, Q(t) \Rightarrow t \in M$, where Q is said to be characterizing *the mode* M .

That is, a mode M is a subset of states that satisfy some predicate Q . Unlike hierarchical state models, a mode itself is not a higher-level state, but it is a group of states. For example, the predicate $(p = norm) \wedge (sb = off)$ characterizes the normal operation mode M_{NORM} in the ESFAS system. M_{NORM} contains all the states where the pressure is normal and the safety blocking sb is not blocked. A state, such as $t = \langle norm, off, inactive \rangle$, satisfies Q_{NORM} and belongs to M_{NORM} . Intuitively, a mode is the endurance of the system operation (or execution) over a set of states, which have a common invariant. We can say that if two identical systems are in the same state, their behavior will be indistinguishable. However, this is not necessarily true for two identical systems in the same mode.

4.3.2 Mode-Class

To be able to model a space T using modes, there must be a number of modes covering the system space. These modes must be disjoint to be usable in an abstract transition system. Such a collection of modes is referred to as a *mode-class*.

Definition 2. (Mode-Class MC). For a space $T = \langle V, D \rangle$, a list of modes $MC = \langle M_1, \dots, M_m \rangle$, characterized by a corresponding list of predicates $Q_{MC} = \langle Q_1, \dots, Q_m \rangle$, is called a Mode-Class MC iff each state $t \in T$ is in exactly one mode $M_i \in MC$. That is, $\forall t \in T, \bigoplus_{i=1, \dots, m} Q_i(t)$.

This means that a mode-class MC is a set of disjoint modes that are covering the system state-space. A MC is a *partitioning* of the space T , induced by the equivalence relationship given by the formula in Definition 2. As an example, we may partition the state-space of the ESFAS system by two mode-classes: the *pressure mode-class* MC^{pr} and the *safety-blocking mode-class* MC^{sb} . The mode-class MC^{pr} partitions the space into three modes M_1^{pr} , M_2^{pr} and M_3^{pr} , which are the modes of the ESFAS computer system, when the plant's pressure is *low*, *norm* and *high*, respectively. Also, the

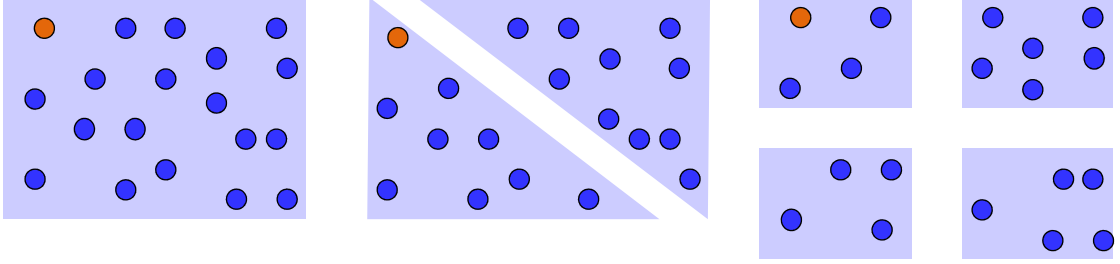


Figure 4.4: Pictorial illustration of state-space partitioning.

mode-class MC^{sb} partitions the space into two modes M_1^{sb} and M_2^{sb} , which are the modes of the ESFAS, when the safety-blocking button is *on* or *off*. The predicate $sb = on$ characterizes the states that belong to the mode M_1^{sb} , and the predicate $sb = off$ characterizes those states belonging to the mode M_2^{sb} . Similar predicates are defined for the modes in MC^{pr} .

As a notational convention, M_i^{class} denotes the i -th mode in the mode-class MC^{class} and Q_i^{class} denotes the predicate characterizing M_i^{class} .

The concepts of *mode-classes* and *state-space partitioning* can be best illustrated via a mental picture of the state-space. Let us imagine the state-space of some system as a rectangle, where its edges are the limits of the state-space and where each space unit on the rectangle is a state. If this system is a continuous system, then the unit of space (i.e., the state) is infinitely small and hence the number of states is infinitely large – so the state-space is a continuum of states.

However, in the case of digital system, the state-space must have a finite number of states, and the rectangle will have a finite number of space points (or space units) – so the state-space is a discrete set of states². In our mental picture, the (discrete) state-space is represented by a set of small circles lying within the perimeter of the rectangle (of the continuous counterpart system). The (discrete) system always exists only in one of this finite number of states, and never exists in any other state in the continuum of this rectangle.

Figure 4.4 spatially depicts the state-space (of some discrete system) as rectangle with the discrete states depicted as small circles.

The figure shows three possible partitionings of the state-space so that the discrete states set is partitioned into subsets of states where each subset corresponds to a mode. Each of these partitionings is a possible mode-class of the system. Note from the figure that in the same partition the states subsets are not overlapping. However, it is possible to find overlaps between two subsets

²Remember that our digital systems are based on the fundamental assumption that, at any point of time, the system exists in one of a finite set of states and that the time the system takes to change state is too short to be observable (otherwise the transition will be itself a state).

that belong to two different partitionings.

We can conclude here that a mode-class is an abstract representation of the system. We can clearly see that, if we hide the states in any of the three partitionings in Figure 4.4, so that the discrete system state-space is then represented by a fewer (and more abstract) number of *state-space units*, each corresponds to a mode. When we model a system-to-be, we need to start from such an abstract view, because we do not yet have enough information to define the detailed states of that system. This abstract view can then be refined as we elaborate the system behavior and add more state information in a process of refining the state-space from the (mode-based) abstract view to a (state-based) concrete one. The next section takes the modes a step further and put them in a automata form.

4.3.3 Mode-Machine

In a mode-class, the disjointness property of modes allows us to specify a transition relation between modes – similar to the transition relation between states in FSM – resulting in an abstract transition system. We refer to such a transition system as a *Mode Machine*.

Definition 3. (Mode Machine MM). For a space $T = \langle V, D \rangle$, a structure $MM = \langle V, Q_{MC}, MC, M_0, \delta, \Delta \rangle$ is called *Mode Machine*, where:

- V is a finite set of system variables.
- Q_{MC} is a collection of predicates characterizing modes in the mode-class MC .
- MC is a mode-class partitioning the space T .
- $M_0 \in MC$ is the initial mode, which includes the initial system state.
- $\delta \subseteq MC \times MC$ is the transition relation. A transition from M_i^a to M_j^a , denoted by $M_i^a \xrightarrow{\ell} M_j^a$ is itself a predicate relation $\ell : V \rightarrow V'$, where the primed variables denote the variables after the transition.
- $\Delta \subseteq V \times D$ is the output function.

The semantics of an MM are that of a standard transition system, with its nodes denoting modes instead of states. Figure 4.5(a) illustrates two mode machines MM^{pr} and MM^{sb} corresponding to the mode-classes MC^{pr} and MC^{sb} , respectively.

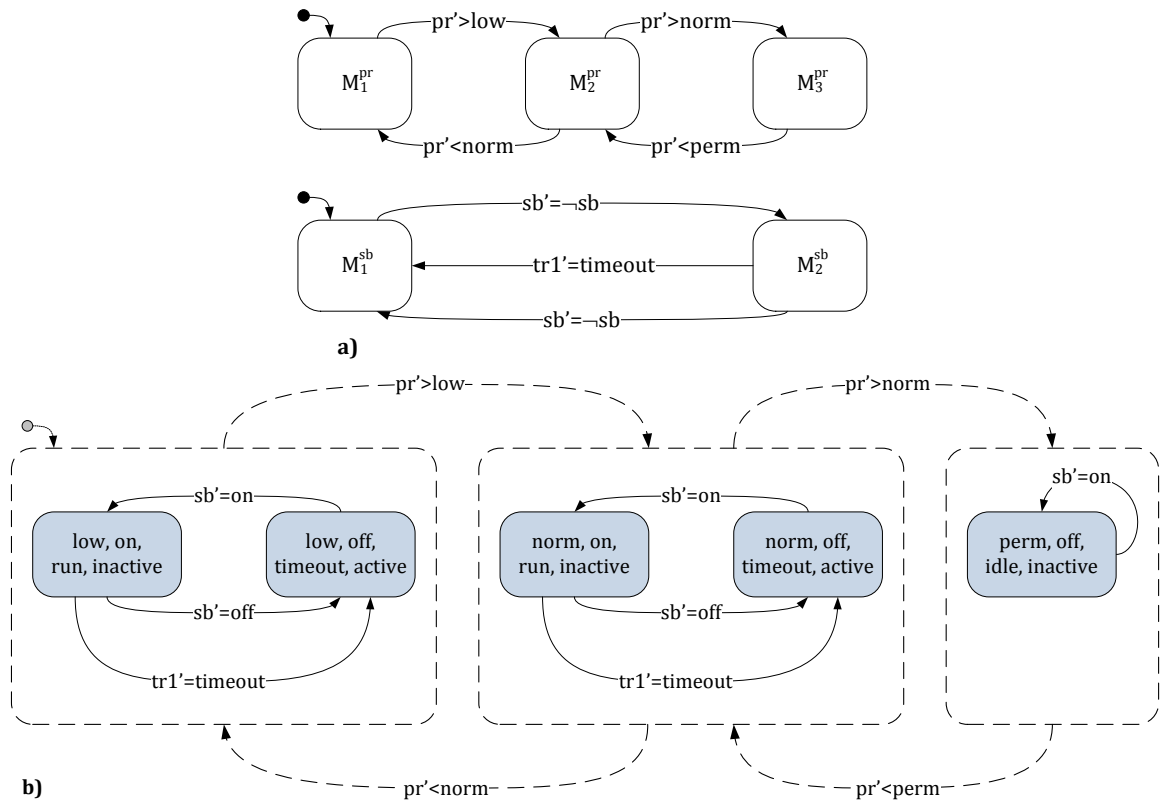


Figure 4.5: (a) Illustration of the ESFAS mode-machines MM^{pr} and MM^{sb} . (b) Illustration of the MM^{pr} mode-machine (in-dotted-lines) refined with the FSMs (converted from the scoped scenarios).

4.3.4 Mode Machine Refinement

We use *mode refinement* and *mode-class refinement* to elaborate the state-space to a more comprehensive description. The ultimate target of refining a mode machine is to reach a *concrete transition system* where all modes are singletons. A mode is *singleton* if

$$\exists!t|Q(t)$$

where Q characterizes this mode. There are two challenges in refining a mode machine: refining the state-space and elaborating the transition relation.

Refining the system's state-space. This requires each mode in a machine to undergo successive iterations of partitioning until we have all modes as singletons, while preserving disjointness and space-coverage. Each mode is partitioned successively into a set of *sub-modes* or *sub-spaces*, constituting a lower-level mode-class which is local to that mode. This is performed in the same sense as we have partitioned the whole system state-space into the very first mode-class.

Lemma 1. (*Mode Refinement*). *Consider a Mode-Machine $MM = \langle V, Q_{MC}, MC, M_0, \delta, \Delta \rangle$. For every non-singleton mode $M_k \in MC$, characterized by a predicate $Q_k \in Q_{MC}$, there exists a local mode-class MC_k that partitions the space part scoped by M_k . MC_k is referred to as refining M_k , denoted as $M_k \prec MC_k$.*

Proof. From Definition 3, the space scoped by M_k is partitioned by MC_k similar to the way the system space is partitioned by MC . □

That is, a partitionable (i. e., non-singleton) mode $M_k \in MC$ can be refined to a class of sub-modes local to M_k and covering only the M_k 's space part. This can be generalized for a mode-class, as follows.

Theorem 1. (*Mode-Class Refinement*). *For a mode-class MC , which is defined on a space T and includes one or more non-singleton modes, there exists a mode-class MC^R , defined on the space T and refining MC , such that the partial order relation $MC \prec MC^R$ holds.*

Proof. Direct from Lemma 1 and Definition 2. □

Theorem 1 can be explained by assuming that MC has only one partitionable mode M_j and all the others are singletons. The mode-class MC is said to be refined by another mode-class MC^R if:

1. there is a subset of modes $MC^j \subset MC^R$ such that MC^j refines the mode M_j . MC^j is said to be a local mode-class to M_j (Lemma 1) because the modes in MC^j are covering the space of M_j only;

2. $MC \setminus M_j = MC^R \setminus MC^j$, which means that the set of modes in MC that are not refined in the $MC \prec MC^R$ relation (i.e., the singletons) appear identically in MC^R .

To reach a concrete model, the state-space should be refined until all modes become states. It is important to note that there may be arbitrarily large possible refinements of the same mode-class. This directly follows from the fact that there may be arbitrarily large numbers of possible mode-classes partitioning the system's state-space.

Elaborating the transition relation. After refining the mode-class, the *refined* modes are no longer part of the model and are replaced by their *refining* modes. As a consequence, the transitions connecting those refined modes must be elaborated, such that they connect the refined model.

Definition 4. (Mode-Transition Elaboration) Consider $MM = \langle V, Q_{MC}, MC, M_0, \delta, \Delta \rangle$ and two modes $M_q, M_r \in MC$ are refined such that $M_q \prec MC^q$ and $M_r \prec MC^r$. A transition is elaborated to a set of transitions $(\forall i, j : M_i^q \xrightarrow{T} M_j^r) \subset (MC^q \times MC^r)$, where $M_i^q \in MC^q, M_j^r \in MC^r$.

We use *space refinement* and *transitions elaboration* as means in our approach to synthesize an integrated system from partial specifications and mode machines. The following sections discuss this process and validate the approach on the Safety Actuation ESFAS case study [36].

4.4 Modes Identification and Selection

One of the main modeling challenges is to identify the basic elements that characterize the system model. A well known example in software engineering is the Object-Oriented development paradigm: identifying the system objects and building their Class-Structure Model is a crucial step for further phases in the development cycle [80, 103, 116]. Unfortunately, there is no standard or systematic or methodical way of doing this task other than using the guidelines-based methods such as the Object-Oriented Analysis and Design OOAD method [80, 116]. Whereas the output artefact of the OOAD method is a structural-model of the system, behavior modeling is concerned with the interactions between those structural elements (resulting in a *separate*³ behavior model for each object, not overall system behavior model) and so the bare OOAD method might not be the ideal tool here.

Our target model is the set of mode-classes and their individual modes, with the semantics described in Section 4.3. As modes are characterized by predicates on system variables, those variables play a crucial role in determining the scope and abstraction-level of modes and mode-classes.

³We remind the reader that the approaches described in this thesis are concerned with the overall system behavior model and do not assume any predefined structure partitioning. However, we believe the approach is equally applicable to the individual system components, considering each component as a system on its own and its environment as the other components interacting with that individual component.

The task of identifying the system variables is analogous to identifying, for example, the system objects in the OOAD method. There must be an initial description, even in textual format, from which the specification task begins. The representation of variables is another issue that largely depends on the application domain. For example, the state of an 8-bit counter can be modeled by a single 8-bit variable, a pair of 4-bit variables, etc. So, the selection of the system variables is application-specific and depends upon the available domain expertise.

Now we turn to the no-less challenging task of identifying the mode-classes and their modes. Modes identification is essentially a task of *architect-ing the system behavior*. At the architecture design stage, designers are interested in the over-arching aspects and they do not know the specific details related to an implementation. By “implementation” here we mean the concrete states and the computations taking place in each state. As with other modeling tasks, there is no standard method for providing an exit criteria for eliciting more (or elaborating the existing) modes. Accumulated experience, though not ideal, is the only viable way to provide heuristics and guidelines to help in this process. In this section we build on our experience of using mode-based specifications and we contribute a technique to help identify mode-classes in a way similar to the formulation of the state-equation of traditional engineering systems (cf. the state-equation model of the bouncing ball in Section 4.1.1). We demonstrate the technique briefly on the Safety Actuation System ESFAS [36]. A detailed demonstration will be provided in Chapters 5 and 6.

4.4.1 Canonical Form of Mode-Classes

Remember the basic definition of *mode* from Section 4.3: *a mode is a set of states, and is characterized by a predicate on system variables*. So, identifying a system mode is a question of what predicate to use to characterize this mode. We would be lucky if the domain experts could provide us with information (even incomplete) on the modes the system is expected to be in. For example, the initial requirements of the Steam Boiler Controller [3] predefine the system’s *Failure Modes* and the (somewhat inconsistent) expected system behavior in each mode. In such cases, the behavior architects would only need to use the system variables and compose the predicates that characterize each of those failure modes. Such a situation is fairly common in control-oriented applications, but, unfortunately, it is rare, or incomplete, in the requirements specifications of general pure-software systems.

In general, we see two approaches to identifying system modes from initial user requirements:

1. The first approach is to intuitively analyze system operations and functional requirements provided by domain experts, and extract the modes information. There could be several

heuristics for doing this:

- (a) one common heuristic applied in hybrid systems [93, 139] is to look for the *steady-state operating points*, where the system is likely to spend considerable time, or where the system performs non-frequent transitions to another operating point. This heuristic is most useful in applications controlling physical processes. If we take our bouncing example, while the ball is in the *falling mode* it falls steadily with constantly increasing speed (the gravity acceleration), it changes this mode only when it hits the ground, and flies up steadily in the *rising mode* repeating itself for a number of bounces.
- (b) Another heuristic is to look for *system configurations*. While in a particular *configuration*, the system shows distinct behavior and functionality. This terminology is pervasive in GUI applications, where the GUI controls (e.g., buttons, check-boxes, etc.) might have very different functions under different configurations (a.k.a. *application settings*.) Text editors are well known examples of these cases.

This approach is *ad hoc* and quite dependent upon the designer's intuition.

2. Another approach is to start from the system variables and predicating directly on them to build the mode-classes behavior model. This approach builds on the assumption that system variables are *definitive* enough to adequately describe the system state-space⁴.

The latter approach is more formal and provides designers with an appropriate tools to partition the state-space with full control on what variables to expose and what variables to hide in each mode-class, assuming that the variables were pre-identified in an earlier modeling step (see discussion in 4.4.1).

We follow the latter approach because it allows for exhaustive exploration of mode-classes (e.g., designers can start modeling a mode-class per variables and combine variables if necessary). The drawback of the first approach is that it could set out of hand very quickly.

In this section, we propose a novel technique for identifying system modes by exploiting a *canonical form* of state variables. We call the resulting mode-classes model a *Canonical Mode-Classes model* as it is inspired by the canonical form of the State-Space Equation model of traditional systems (see Section 4.4.1.1). The questions here are: what are the possible techniques to do this, and what criteria to use to decide evaluate if the technique is effective, practical and scalable?

⁴Other terminology refers to the state variables as *domain variables* that define a *domain theory* of the system.

We attempt to answer these questions in this section. First, we recall our bouncing ball example (Section 4.4.1.1) to showcase our approach for identifying modes by deriving the Canonical Mode-Classes behavior model of the bouncing ball from its canonical State-Space Equation. Even though we derive three mode-classes in this mode, a single mode-class is sufficient and we will discuss why other mode-classes are redundant in this example. Next we take a further step in Section 4.4.1.2 and apply our approach to a real-world system: the Safety-Actuation system ESFAS [36]. We finally conclude with notes and observations in Section 4.5.

4.4.1.1 The *bouncing ball* behavior Mode-Classes

Let us start from our example of bouncing ball system. Equations 4.1 and 4.2 are the State-Space Equations, in non-canonical form, involving two system variables: the ball position p and its velocity v and the trajectories of these two variables are illustrated in Figure 4.1. The canonical form of the State-Space Equations is expressed as:

$$x_1(t) = a \cdot F(t) + b \tag{4.3}$$

$$x_2(t) = \dot{x}_1(t) \tag{4.4}$$

$$x_3(t) = \dot{x}_2(t) \tag{4.5}$$

where $F(t)$ is the parabolic function of the bouncing trajectory of an arbitrary bounce (see Figure 4.1). The variables coefficients, a and b , are variables represent the ball's material. The motion of the bouncing ball is described by three state variables; its position x_1 , its vertical velocity x_2 and its acceleration x_3 .

Having established the State-Space Equation in its canonical form, the Canonical Mode-Classes model can be derived from these state equations such that, for each state variable, the range of the variable's function is partitioned into a set of sub-ranges that identify a corresponding set of modes in a one-to-one relation. This set of modes constitute a mode-class.

Now let us apply this procedure to the bouncing ball example. The trajectory of the position function $x_1(t)$ can be partitioned into two ranges corresponding to the falling and rising sub-trajectories, which are easily described via parabolic functions, (see Figure4.1). The partitioned version of the position trajectory function is now expressed as:

$$x_1(t) = \begin{cases} a.F_{rise}(t) + b & \text{during the rising mode} \\ a.F_{fall}(t) + b & \text{during the falling mode} \end{cases} \quad (4.6)$$

where $F_{rise}(t)$ and $F_{fall}(t)$ refer to the functions of, respectively, the rising and falling trajectories of the overall parabolic trajectory of an arbitrary bounce (as in Figure 4.1). This partitioning of the function variable $x_1(t)$ results in a mode-class MC_p that has two modes: the *raising mode* M_{rise}^p and the *falling mode* M_{fall}^p , where $M_{rise}^p \equiv a.F_{rise}(t) + b$ and $M_{fall}^p \equiv a.F_{fall}(t) + b$. Using the notation in DEFINITION 4, we refer to the position mode-class $MC_p = \langle M_{fall}^p, M_{rise}^p \rangle$.

Obviously, a similar partitioning can be made on the trajectory functions of the ball's velocity $x_2(t)$ and acceleration $x_3(t)$ variables. The partitioned versions of these state variables are expressed as follows:

$$x_2(t) = \begin{cases} -\dot{x}_1(t) & \text{during the slowdown mode} \\ \dot{x}_1(t) & \text{during the speedup mode} \end{cases} \quad (4.7)$$

$$x_3(t) = \begin{cases} -9.81 & \text{during the deceleration mode} \\ 9.81 & \text{during the acceleration mode} \end{cases} \quad (4.8)$$

As in the case of the position state variable p , the partitioning of the velocity and acceleration state variables results in two additional mode-classes: velocity mode-class $MC_v = \langle M_{speedup}^v, M_{slowdown}^v \rangle$ and acceleration mode-class $MC_a = \langle M_{decel}^a, M_{accel}^a \rangle$, where $M_{slowdown}^v \equiv \dot{x}_1(t)$, $M_{speedup}^v \equiv -\dot{x}_1(t)$, $M_{decel}^a \equiv -9.81$ and $M_{accel}^a \equiv 9.81$. The corresponding mode-machines are drawn in Figure 4.6.

We could easily observe that these mode-classes coincide, such that the modes M_{rise}^p , $M_{slowdown}^v$ and M_{decel}^a completely overlap, and similarly the modes M_{fall}^p , $M_{speedup}^v$ and M_{accel}^a overlap. This means that a single mode-class would suffice to describe the discrete part of the ball's motion. Formally speaking, if we apply the Theorem 1 (mode-class refinement) to these mode-classes we will find that the resultant mode-class is equivalent to any of the merged classes. Figure 4.7 illustrates the result of this merge⁵. In this figure, the derived mode-classes in Figure 4.6 are merged in one mode-class we labeled as $MC_{BallMotion}$, where the modes M_{rise}^p , $M_{slowdown}^v$ and M_{decel}^a are merged in one mode $M_{up}^{BallMotion}$ and the modes M_{fall}^p , $M_{speedup}^v$ and M_{accel}^a are merged in one mode $M_{down}^{BallMotion}$.

⁵Note that, in Figure 4.7, we arbitrarily used the variable x_2 in mode transition, however, the events on the other state variables, in Figure 4.6, could also be used because all of these events are simultaneous and cause the same transitions.

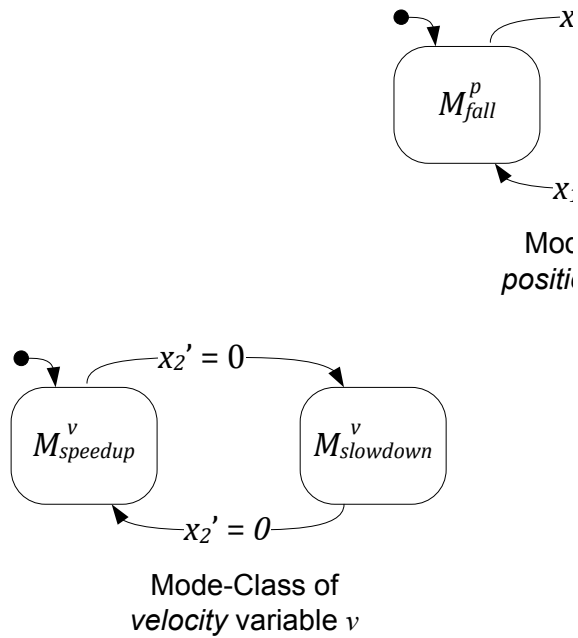


Figure 4.6: Bouncing ball mode-classes

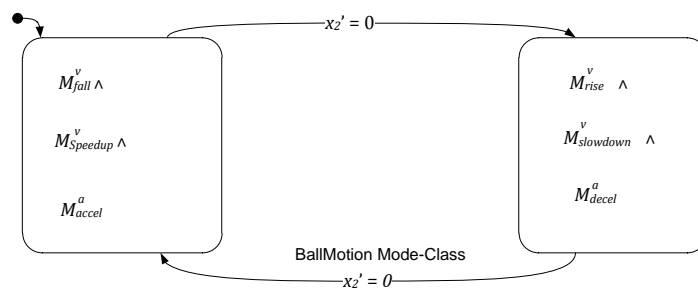


Figure 4.7: Bouncing ball's position mode-class $MC_{BallMotion} = \langle M_{up}^{BallMotion}, M_{down}^{BallMotion} \rangle$

Table 4.1: State variables of the ESFAS controller

variable name	description	domain values
pr	steam pressure	HIGH, NORM (i. e., normal) or HIGH
sb	pushbutton to block the safety signal activation	ON or OFF
ss	safety-signal status	ACTIVE or INACTIVE

The bouncing ball example is a rather simple example of a hybrid system. Although simple, the bouncing ball example demonstrates an important aspect of traditional systems: the behavior regularity. One apparent regularity is the symmetry of the trajectories in the two modes of each state variable – the trajectory functions differ only in sign. Another interesting, but somewhat implicit, regularity here is the exact *synchronization* between the transition points in all mode-classes of the state variables. The mode-classes in Figure 4.6 coincide, because the state variables have strong and regular dependencies, such that they synchronize together (i.e., they change at the same time and towards the same modes). This is not the case with systems having more discrete, requirements-driven behaviors. In the following section we will examine a more useful example, the ESFAS Controller [36], and derive its Canonical Mode-Classes model.

4.4.1.2 Canonical Mode-Classes of the ESFAS Controller

The ESFAS computer controller is part of power plant and is intended to mitigate damage to the plant when faults occurrence. ESFAS receives signals from different sensors and checks if the signal level has reached predetermined set-points, in which case ESFAS sends a safety notification to another safety-handler system which deals with the accident [36]. We identified in Table 4.1 the state variables⁶ necessary to define the ESFAS state-space. Also Figure 4.8 visually illustrates the canonical mode-classes model of the ESFAS controller.

Notice that each mode-class in Figure 4.8 is concerned only with its corresponding variable with abstraction of all other variables. So, each mode-class represents a concern and is valid at this abstraction level. We can deduce more detailed (and meaningful) mode-classes by (incrementally) merging these mode-classes together. Figure 4.9 illustrate a merge of all ESFAS canonical mode-classes in a single detailed mode-class.

In theory, the result of merging the mode-classes could be twelve modes (the cross product of all modes in all classes). However, due to the system’s constraints (e.g., safety can not be blocked if

⁶Note: these variables are slightly different from the original problem specifications for brevity and clarity purposes. For example, the original specifications assumes another pushbutton RESET in addition to the safety-signal blocking pushbutton BLOCK, however we assume the BLOCK button performs both functions. This emphasizes the conjecture that variables identifications might well vary from design to another.

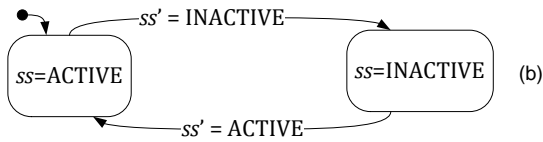
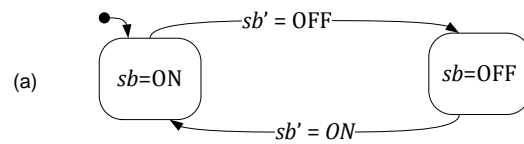
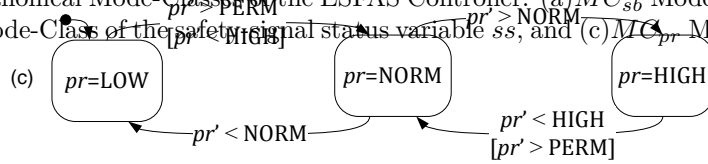


Figure 4.8: Canonical Mode-Classes of the ESFAS Controller: (a) MC_{sb} Mode-Class of the pushbutton sb , (b) MC_{ss} Mode-Class of the safety signal status variable ss , and (c) MC_{pr} Mode-Class of the pressure variable pr .



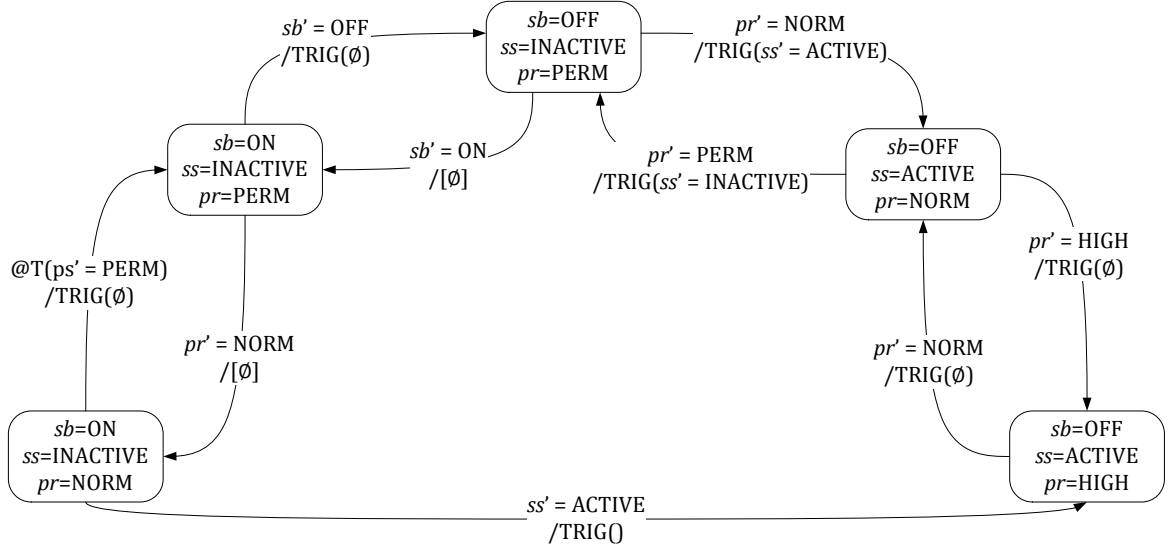


Figure 4.9: ESFAS mode-classes after merging and refinement to a state-level model.

pressure is high), only five modes are possible, as in Figure 4.9, and the rest of the modes are removed from the system's state-space (because they are not possible). These (five) modes are now considered as concrete states, as in Figure 4.9, if no further refinements (i.e., more state variables) are identified in the specifications. But, of course, they are still *modes* with respect to any implementation (e.g., in some computer language) that might introduce implementation-specific intermediate variables due to, for example, the word size of the target computer.

Note that in Figure 4.9, we use the predicate $\text{TRIG}(event)$ to indicate an event is triggered. This follows the standard format used in Statecharts.

The merging of mode-classes passes through several successive steps until we reach a concrete system model, such as the one in Figure 4.9. Though it looks a simple and a small example, the synthesis of the ESFAS behavior model in Figure 4.9 is a non-trivial task. We will revisit this process in detail in Chapter 6.

When we consider larger systems that has too many state variables, it obviously will be impractical to derive a mode-class for each variable. Indeed, starting the behavior architecture with too large number of mode-machines will frustrate this goal and the target (automata-based) behavior model will get complicated too quickly when we begin to merge the first few mode-machines.

Recall that the original goal of our work is to research methodologies and techniques to simplify and facilitate the behavior modeling task. So, to support our Canonical Mode-Classes modeling technique, we need to augment it with some practical guidelines to help designers to select a subset of

state variables to be used in the Canonical Mode-Classes model. Besides the need for simplifying the modeling process, this augmentation is also motivated by the fact that, for an arbitrary system, not all variables will be useful in the Canonical Mode-Classes. So, the modeling process has to select among the state variables, those that are candidates for modeling the mode-classes, i.e., those variables that *drive* the modal aspect of the behavior. In the following section, we propose an intuitive heuristic for deciding on those variables that drive the modality of the system.

4.4.2 Modal vs Operational Behavior

The main rationale underlying the concept of *modes* is the behavior's irregularity that results from mixing independent *aspects* or *concerns* in the same view. Identifying those "regular" behaviors and classifying each group of them as a mode-class provides a mechanism to separate them so that each view is comprehensible. The question we discuss in this section is how to distinguish between the variables that drive modality in behavior and those that do not.

There is no black-and-white method for doing this because the definition and modeling of variables depend on both the application and the designer's experience – analogous to popular Software Engineering heuristics such as OOAD (see section 4.4). Our heuristic approach to make this distinction is a trial-and-error categorization of the variables into those that drive *Modal Behavior* and others whose processing represents the *Operational Behavior* of the system.

By *Modal Behavior* we mean the occurrence and sequencing of conditions that trigger a transition of the system from one function to another. We express these conditions as predicates on system variables and we call these variables *Modal Variables*. As we discussed in Section 4.3, those predicates characterize modes. The conditions Predicates and their sequencing (i.e., mode-transitions) collectively constitute mode-machines. We refer the reader to Section 4.3 for more formal definitions and details.

By *Operational Behavior* we mean the processing performed on state variables (and possibly intermediate variables) to perform a certain function that drives one or more outputs of the system. At any step in this processing, one or more Modal Variables might change, and in turn the function being performed is exited (due to a mode-transition) and a different function started.

This model of computation is standard in modern and classical computing literature (e.g., Dijkstra's thesis of *pre-* and *post- conditions* as specifications of programs). We attempt to reuse it here as a basic tool in this task of behavior modeling in terms of modes.

4.5 Discussion and Summary

In this chapter, we elaborated on the concepts of *modes* and *mode-classes*, which are central to the work in this thesis. We gave a background to the origins of *modes* in Hybrid Dynamic Systems [93, 139] and their emergence in software engineering [7, 63]. We made a clear distinction between the idea of *modes* used in this thesis, and the term *modal* frequently used in computer science literature, such as Modal Logic [30], Modal Transition Systems [87], etc. We built on the original idea of *modes* [63, 7] and provided insights into the potential of *mode-classes* to serve as multi-dimensional (such as [130]) models of software behaviors to achieve N-Degree behavior models.

This chapter also provided a sound formalization of *modes* and *mode-classes* using simple concepts from standard predicate logic. We defined the abstract, state-based model called *Mode-Machine*, which we will use throughly in Chapter 6, where we describe our synthesis approach.

We also provided the technique to identify a reference model of mode-classes, which we call the *canonical mode-classes*, given a defined set of state variables. The *canonical mode-classes* model is inspired by the *canonical form* of the *state equation* well known in Hybrid Systems [93, 139]. This model is a reference model and we presented it for illustrative purposes; however, in our experience it is a good starting point. So, designers shall decide on the appropriate mode-classes that help in describing the system behavior according to the application domain.

Some of system variables may be involved in mode-based specifications, but not all. This is because the modeling task will become complicated quickly if we put everything in automata (i.e., modes predicate). The main goal of the thesis is to develop an approach to facilitate modeling from multiple forms of specifications. So, the distinction between modal and non-modal variables helps, on the one hand, to leave a room for other intuitive specifications such as interaction scenarios. This helps greatly to reduce the complexity of automata-based modeling while still keeping its expressive power. On the other hand, it strips the modal actions/events from the operational scenarios and so it guide the designers in the task of specifying scenarios.

Both mode-based and scenario-based specifications are specifying behaviors. The challenge of specifying both types of behaviors is a question of which behaviors should go in modes and which behaviors should go in scenarios. In the next chapter we will provide an answer to this question by clarifying the behavior types that are suitable for each case.

Chapter 5

Structured Scenario Specifications

Separation of concerns [105] is a provably useful concept to support modularization of software specifications and design. In this chapter, we propose to apply the separation of concerns principle to structure scenario-based specifications, in an attempt to address the current issues of scalability and maintainability in scenario-based specifications. As a particular interest of this thesis, the structuring of scenario specifications also facilitates the application of synthesis approaches to generate automata-based design model (see Chapter 6) suitable for further system analysis and code generation.

In Chapter 4, we elaborated on and formalized the concepts of “mode” and “mode-class” [7, 63] as behavior abstractions. In this chapter, we use these concepts to introduce separation of concerns in scenarios in two complementary ways:

1. Firstly, mode-class partitions the system’s state-space from a certain perspective, and designers specify scenarios *concerned with* this aspect. Having several mode-classes allows for separating the different system concerns, so that each mode-class addresses one of them (see Chapter 4 for detail). In this sense, scenarios will be structured to address different concerns, thanks to mode-classes.
2. Secondly, the modes belonging to the same mode-class establish separate *contexts*; each mode represents a *context*. By *context* here we mean the condition(s) prescribed by the characterizing predicate of the mode associated with this context (see Chapter 4 for foundations of the *characterizing predicate* of a mode). As we will see shortly, Scenarios are specified under the scope of (or within the context associated with) each mode and no scenario is allowed to cross the boundaries of its scoping mode. In this sense, a behavior in a scenario will be focused in a certain context, leaving the inter-contexts behaviors to the transitions between modes in the

mode-class.

In this chapter we examine how effective the concept of *modes* is as a means for separating concerns in scenarios and structuring them for improved elaboration and elicitation. Next, in Chapter 6, we will demonstrate a further benefit of scenarios structuring where we propose the approach of automata-based model synthesis from structured scenarios.

5.1 Introduction

Scenarios are the most pervasive form of specifications can be found in current requirements documents. Popular dialects of scenarios are *message sequence charts* (MSC) [69] and UML *sequence diagrams* SD [103] (see Chapter 2 for a detailed background). Scenarios are meant to specify the expected interactions between the system and its environment and (or) interactions among the system's components when a decomposition of the system is already in place. The intuitive presentation of scenarios facilitates stakeholder involvement in the specifications task.

In their flat structure, scenarios do not scale up with system complexity, and they tend to get too complex quickly. Non-trivial systems may involve several components with a large number of (typically irregular or non-patterned) interactions between these components. With the possibility of several alternative *flows* (i.e., sequences of steps) in the same scenario, designers have little or no guidance on which way to go in the scenario flow, and they resort to using a combination of programmatic constructs, such as **alt** in MSCs (See Chapter 2), which accelerate the tendency of the scenarios to get too complex to specify a meaningful requirement. Moreover, there is no guidance on when and where (i.e., at which conditions or state) to start a scenario and as a result designers try to (randomly) add conditions (using, for e.g., the MSC condition constructs) to express the initial conditions of the scenario. It looks like we need to *setup a context* for the scenario, but unfortunately it is done in a *ad hoc* way. A similar confusion arises when we decide when the scenario should be stopped.

Traditionally, designers start from an initial state and keep on adding more steps until they run out of possible steps to add; “why should I stop when there are still more possible steps to add!” the designer thinks. This will result in lengthy and unmaintainable scenarios. Moreover, what if we wanted to specify steps for an alternative scenario, how would we explicitly distinguish between these two scenarios without having to go through the step flows to spot the different steps?

It is not surprising that these issues are inherent characteristics of scenarios as a specifications artefact. Indeed, a scenario (as commonly perceived [27, 28, 55]) is just one of several possible executions of the system, which does not necessarily cover all the behaviors. A scenario does not

indicate when this execution is triggered and when it ends. Previous attempts in [123] and [38] mandates a “preamble” of interactions for a scenario, as a trigger for its execution, however, this “preamble” is still specified in terms of interactions, and further it does not help to setup a context for scenario execution. Another specification problem with traditional scenario specifications is that a scenario does not say anything about its relationship with other, independently elicited, scenarios. The *Gates* construct, in MSCs, tells us only about the outgoing (or incoming) messages from (or to) the current MSC scenario, but is silent about with “whom” these messages are exchanged, and how they interleave on the lifelines of the other scenarios. The *References* feature is another construct in MSC that relates a sub-scenario to its parent scenario (see Chapter 2), however it serves only to compact the pictorial view of a long scenario.

These issues are a natural consequence of the lack of proper mechanisms to structure scenario specifications and their inability to facilitate separation of crosscutting concerns. The undisciplined specification of scenarios at the early stages of development has led to *ad hoc* elicitation of the possible behaviors – typically elicited as direct extraction from a given textual description – and so the coverage of the behavior space depends on human comprehension which falls short for nontrivial systems. Unfortunately, this impedes the chances of proper elicitation of requirements and, consequently, increases the partiality problem inherent in scenario-based specifications.

Extensive research efforts have been devoted to improve the usability and scalability of scenarios, most notably the ITU Telecom standard of MSC [69] and its extensions (such as HMSC [72] and LSC [38]). Also, the recently introduced interaction overview diagrams, IOD [103] in UML 2.0, is an attempt to compose scenario-based specifications. There are two common observations on these models. First they compose existing scenarios and do not provide means for elaboration on or eliciting new scenarios. Second, they compose scenarios in flowchart-like structures with subjective relationships between scenarios, such as causality, timing, etc. Even though concurrency and causal relationships are considered, it is not helpful to use a flowcharting structure of scenarios to comprehend the boundaries of the system behavior-space and its possible executions [104].

In this chapter we are particularly concerned with the abovementioned issues. We propose a novel approach to structure scenario-based specifications by providing a behavior context for each scenario, or for a cohesive set of those scenarios. These behavior contexts are specified through a disciplined approach of partitioning the state-space from different (crosscutting) viewpoints. Our approach can be seen as setting up the proper contexts that guide the scenario specification task. The designer can then safely think of scenarios that fit in each context without having to worry about the whole behavior space. In this way, we provide a predictable process of specifying scenarios. Moreover, the

state-space partitioning helps to keep track of the boundaries of the behavior space, allowing for further elaboration of the specified contexts and/or scenarios, in addition to the possibility of uncovering behavior gaps for which more behaviors are elicited.

The rest of this chapter is organized as follows: Section 5.2 provides an overview of methods concerned with the complexity issues in scenario specifications; Section 5.3 describes our modes-based approach to structure scenario specifications; and in Section 5.4, we apply this modes-based approach to two case studies.

5.2 Scenarios Structuring: State-of-the-Art

To obtain an understanding of structuring scenario specifications, in this section we discuss the current trends and techniques for composing scenarios, focusing in particular on modularization. We identify potential improvements, in terms of separation-of-concerns, in the current techniques. More concretely, we pinpoint their inability to compose scenarios and structure them in the system's state-space, rather than composing scenarios in a flowcharting manner.

In this section, we first survey the current techniques for scenarios composition and structuring. In Section 5.3 we describe a novel method of structuring scenarios, using the specification framework we described in Chapter 4. Briefly, we partition the state-space (from different viewpoints) into contexts and organize the different scenarios within those contexts in such a way that facilitates uncovering gaps in the state-space. This triggers the elicitation of more behaviors (i.e., scenarios), which is very useful in the early stages of development. We demonstrate these advantages with case studies later in this chapter.

One of the richest forms of scenario specifications is the Telecom standard of message sequence charts (MSC). MSC-96 is the original version of the MSC standard, but the standard has been extended further in MSC version 2000. We devoted Chapter 2 to a detailed analytic discussion of MSC-96 as role-model of basic scenario specifications.

Hierarchical Message sequence Charts (HMSC), part of the ITU standard [72], provide structuring techniques to compose more complex specifications from basic MSCs. Another standardized approach is Interactions Overview Diagrams (IOD) recently introduced in UML 2.0. IODs use quite similar techniques to the ones used in HMSC but with additional features, such as 'preemption' and 'suspension' in scenarios. Apparently, the features in IODs have very close one-to-one correspondence with Statecharts [59, 61] and seem to be designed for straightforward translations to Statechart machines. This might restrict IODs from being used in other settings.

Most of the existing scenarios specification methods employ quite similar techniques to combine small or “basic” scenarios into more composite structures that are informal, and, when visualized, are close to flowcharts diagrams. However, to the best of our knowledge, there is a complete absence of proposals that promote structuring of the behavior space – at its outset – before writing the scenarios themselves.

The presentation in this section is neither comprehensive nor overly critical of the design choices made to structure scenario-based specifications. The reason is that our proposed structuring technique, presented later in this chapter, combines two complementary types of specifications (namely, modes and scenarios), and thus it takes a “lateral” direction compared to the existing approaches.

5.2.1 High Level Message Sequence Charts HMSC

The syntactic constructs of MSC-96 [69], presented in Chapter 2, allow us to specify both simple and compound sequences of interactions. There are also additional means in MSC-96 for *Referencing* mechanism we can use to decompose large interaction patterns into manageable parts (see [69, 119] for more details). This is a first step towards increasing the comprehensibility of large MSC specifications. It does not suffice, however, for conveying the “big picture”, i.e., the way all the MSCs that form a specification are related or composed; we still have to find and follow all references within an MSC document to obtain the sequences of interactions occurring in the system under specification. In addition, the instance axes appearing in all MSCs add to the complexity of the pictures we draw. If we wish to represent, say, three successive phases of a communications protocol as connect, transmit, and disconnect, we do not want to reveal right from the beginning of the development process into what components the system decomposes, and what the exact interactions among these components are. Instead, we would like to say something like “an execution of the system consists of an infinite sequence of steps; each step consists of three consecutive phases: connect, transmit, and disconnect”. This high level specification of the protocol references neither components, nor messages. However, none of the syntactic elements we have studied up to now allows us to specify interaction sequences at this high level of abstraction. MSC-96 introduces High Level MSCs (HMSCs [72]) as a notational alternative to the plain MSCs, to address these problems. We discuss these features in this section.

An HMSC is, in essence, a directed graph, whose nodes reference (H)MSCs; the graph describes a “roadmap” through the MSCs referenced in the nodes. An edge from a node labeled MSC X to a node labeled MSC Y in the graph, indicates that part of the interactive behavior of the system consists of a sequencing of the interactions specified in X and those specified in Y. The edges determine how we must “paste together” the MSCs referenced in the nodes to obtain the interaction sequences of the

system.

More precisely, each node of an HMSC is any one of the following

- a start node,
- an end node,
- a reference node,
- a parallel node,
- a connection node,
- a condition node.

We will introduce each of these node classes informally. We also briefly relate HMSCs to plain MSCs, so that we can translate the former into the latter.

5.2.1.1 Start, End, and Reference Nodes in HMSC

Each HMSC has exactly one start node, whose graphic representation is a downward outlined triangle. The start node indicates the beginning of any interactive sequence we can derive from the HMSC: it has no incoming edges. An end node, whose graphic representation is the horizontal mirror image of the start symbol, terminates paths through the HMSC; it has no outgoing edges. Interaction sequences obtained by following any path through the HMSC will end when we reach an end node. Reference nodes are similar to the MSC reference symbols we have already discussed. Their labels may be MSC reference expressions as before; they also have the same graphic representation.

Consider the HMSC *sequential* in Figure 5.1. It consists of a start node, two MSC reference nodes, *OverrideSafety* and *PressureChange*, and the end node. The start and end nodes in HMSC are depicted as small blank triangles directed down (for start node) and up (for end node). The arrows indicate valid paths through the graph. In this case there is exactly one path through the HMSC, and the basic MSCs *OverrideSafety* and *PressureChange* are then executed sequentially. The execution begins at the start node, passes through the MSC references for *OverrideSafety* and *PressureChange*, and then stops at the end node. Intuitively, we obtain the semantics of this HMSC by pasting the interactions within the MSC *OverrideSafety*, in their specified order, over those within the MSC *PressureChange*, and by determining the resulting MSC's semantics according to the event ordering rules introduced in Chapter 2.

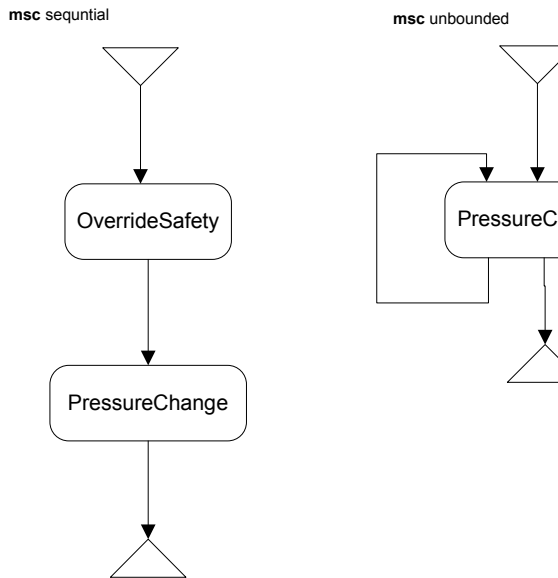


Figure 5.1: Examples of composition structures in HMSC: sequential, unbounded repetition and alternative

The composition in the HMSC *sequential* is known in MSC-96 as Weak Sequential Composition. Assume we are given an arbitrary HMSC with an arrow from a reference node labeled A to a reference node labeled B. In the composite HMSC there is a semantic difference between events on common instances of A and B, and events on instances appearing in only one of the two referenced MSCs. On common instances, A's events precede B's events. Events on instances of A that differ from B's instances are independent of events in B. Similarly, events on instances of B that are not also instances of A are independent of events in A.

The start node, and all reference, parallel, condition, and connection nodes of an HMSC can have an arbitrary number of outgoing edges. In fact, all nodes, except the end node, must have at least one outgoing edge. A node with more than one outgoing edge indicates the existence of an alternative path through the HMSC. For example, in the HMSC *alternative* in Figure 5.1, the forking of flow after the HMSC start node indicates exclusively alternative (but not parallel) paths. Intuitively, the semantics of the *alternative* HMSC is the set of interaction sequences obtained by following either of the two possible paths through the graph. Note the similarity of this construct and the **alt** expression in basic MSC (see Chapter 2).

MSC-96 allows cycles in HMSC graphs. This corresponds to unbounded repetition of the interactions determined by the nodes along the path forming the cycle. The *unbounded* HMSC in

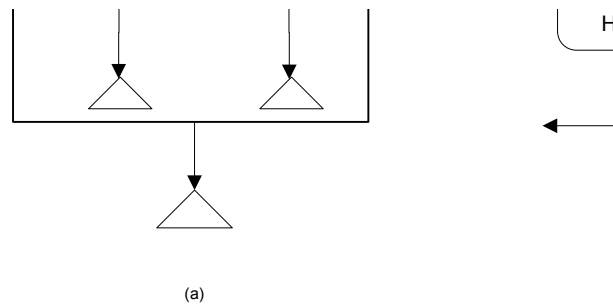


Figure 5.2: (a) Example of parallel composition HMSC. (b) illustration of Connections Nodes in HSMC

Figure 5.1 is an example of an unbounded cycle in a HMSC graph. Intuitively, its semantics consist of an unbounded (but finite) sequencing of the interactions represented by the HMSC PressureChange. Note the similarity of this construct and the **loop**<1, inf > expression in Basic MSC. Infinite cycles are also provided in the HMSC standard. The *unbounded* HMSC graph can be converted to infinite repetition if we remove the end node; this corresponds to the inline expression **loop**<1, inf > in MSC-96.

5.2.1.2 Parallel Composition in HMSC

Besides sequential composition, alternatives, and repetition, HMSCs also allow us to specify independence of the events of entire MSCs. For that purpose, MSC-96 introduces the parallel node. Its graphic representation is a box, as illustrated by the example in Figure 5.2. A parallel node may contain any number of HMSCs. Intuitively, the events occurring in the HMSCs within a parallel node are mutually independent. As with the other composition structures discussed above, there is a correspondence between the parallel composition mechanism in HMSC and the **par** inline expression in MSC-96.

5.2.1.3 Connection Nodes and Condition Nodes in HMSC

Edges in HMSCs may only connect nodes. To facilitate readability of HMSCs, MSC-96 introduces connection nodes, whose sole purpose is to form the starting or ending point of edges within the graph. This helps to reduce the number of incoming and outgoing edges, in particular, of reference nodes. The graphic representation of a connection node is a non-filled circle.

A condition node, graphically represented exactly as the condition symbol in simple MSCs, restricts the MSCs that may precede and succeed it in an HMSC. The ITU-96 standard [69] defines a plethora of corresponding requirements. The semantics definition in ITU-98 [70], however, assigns no meaning whatsoever to condition nodes. A much simplified (and thus nonstandard) version of the restrictions stated in [69], applicable to the combination of a condition node and a reference node whose label is an MSC name, has two constituents:

1. If there is an arrow from a condition node labeled C to a reference node labeled X, then X must start with a global condition labeled C;
2. If there is an arrow from a reference node labeled X to a condition node labeled C, then X must end with a global condition labeled C.

We refer the reader to the literature mentioned above for the full-fledged set of restriction rules.

5.2.1.4 Summary and Discussion

Hierarchically composing scenarios is an intuitive technique for building compound specifications. The classic term “hierarchy” is somewhat ambiguous and it is not enough to define a relation between the composed entities [105]. In HMSCs, scenarios are composed “hierarchically”, based on the *execution flow* relations such as branching, serial and parallel flows. So, HMSC composition looks like like a flowchart of scenarios. Flowcharting techniques have been proven to be naive, if used at this level of abstraction [104], though they are popular at the detailed level of programming. In general, HMSCs can be good as an initial artefact, useful, for example, to communicate to non-technical stakeholders, but we would not chose them as an elaborate specification input to further analysis or synthesis. In the next section we discuss IOD which is a more elaborate approach for scenarios composition.

5.2.2 UML Interaction Overview Diagrams IOD

UML *release 2.0* [103] introduced Interaction Overview Diagrams (IODs), a notation based on activity diagrams, for specifying relationships between basic interactions sequences (i.e., UML

Sequence Diagrams SD). IODs are based on HMSCs [72]. IODs are a graph-oriented way of specifying relationships between UML interaction diagrams.

Similar to the ITU MSCs, the UML standard interprets the semantics of messages in scenarios, in terms of send and receive events corresponding to the sending or receipt of a message by a participant. Events are partially ordered by weak sequential composition that defines the following: (1) events on the same lifeline are ordered according to their vertical location; (2) a message's send event always comes before its receive event. This is pretty similar to those semantics of MSC-96, with weaker ordering in MCS-96.

UML2.0 introduced *interaction fragments* as a way of increasing the expressiveness of sequence diagrams. An interaction fragment is a box defining a subset of the messages in a sequence diagram and stating that the messages within the fragment are optional, alternatives to each other, can execute in parallel, or should not be allowed. Each fragment is defined by an operator, which states the connection between the fragment's messages and one or more operands. Examples of these operators are **alt**, **opt**, **par**, **neg** and **seq**. The operators **opt** and **neg** have one operand and define that the operand's messages are optional or should not be allowed, respectively. The operators **alt** and **par** have two or more operands and define that the messages within each of the operands are either alternatives or can run in parallel, respectively. The operator **seq** is used to explicitly mark that a sequence of messages is joined by weak sequential composition (the default in UML).

Each node in the activity graph, Figure 5.3, is a reference to an interaction and the edges between activity nodes allow the definition of relationships between interactions, such as parallel execution, alternatives, and control flow. The references in Figure 5.3-(a) are to sequence diagrams, although the references could also be to communication diagrams or other interaction overview diagrams.

UML includes state invariants as a way of defining states in sequence diagrams. State invariants are essentially labels that can be used to identify different points along the lifeline of a participant (or participants). Before their introduction in UML 2.0, they were known as *state labels* in the literature. State invariants are useful to capture the fact that two points along a lifeline are meant to be the same. More concretely, all the events or messages that appear after some state, and before the next state (if any), are to take place while the system is in that state preceding those messages or events. Similar to the Condition Nodes on MSC-96 and HMSC, a State Invariant in an Interaction Fragment which is allowed to span many (or all) objects.

An IOD is actually a restricted form of UML Activity Diagram for capturing some kinds of relationships between interactions. Except for the activity nodes the other notation elements for interaction overview diagrams are the same as for activity diagrams, such as initial, final, decision,

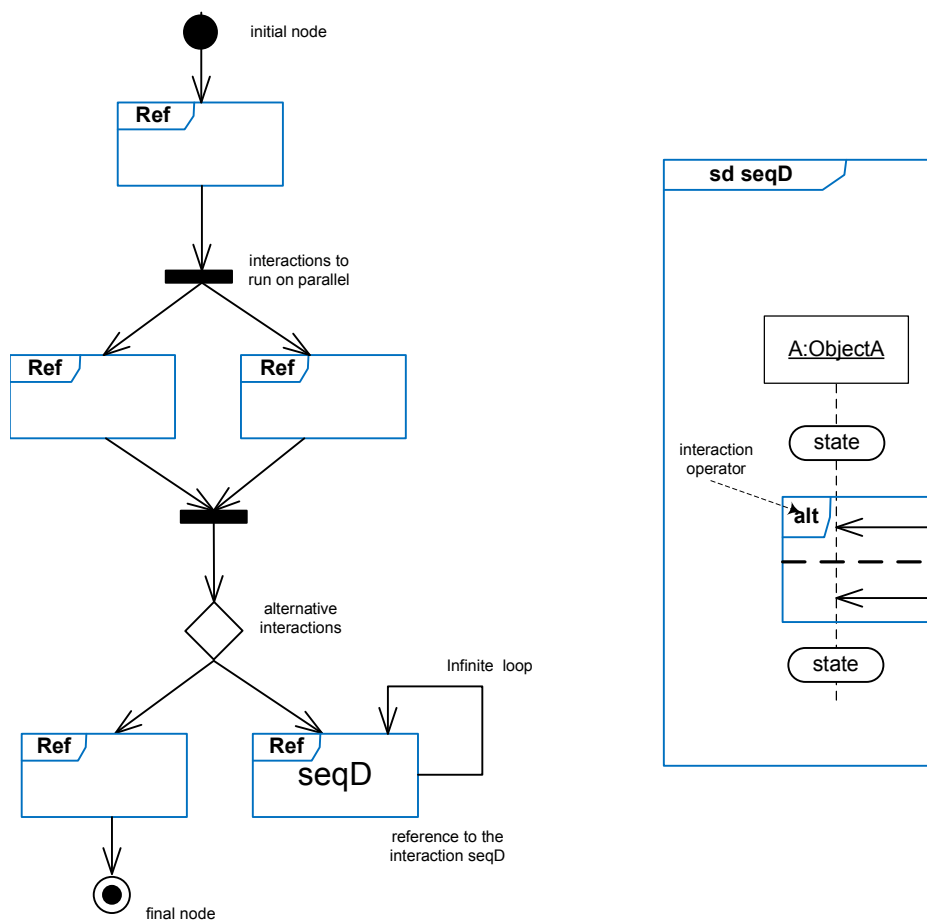


Figure 5.3: UML Interactive Overview Diagram features: Activity Diagram (left) and basic Sequence Diagram SD (right)

merge, fork and join nodes. The two new elements in the interaction overview diagrams are the 'interaction occurrences' and 'interaction elements'.

IOD focus on features such as *forking*, *preemption* and *suspension*, which have close one-to-one correspondence with same features in Statecharts [59, 61]. This intimate relationship between the features in Statecharts and IODs might restrict the application of IODs from being translated to automata-based models other than Statecharts.

5.2.3 Early Aspects

The Aspect Oriented Software Development (AOSD) paradigm¹ puts forward the concept of aspect to modularize crosscutting concerns. AOSD has been successful in introducing methods, tools and programming languages to support development at the code and implementation levels, but it provided little or no support to development activities taking place at earlier stages of development, such as requirements and architecture. “*It is not surprising that the majority of the [AOSD] results focus on the use of programming technologies; only one paper [in [76]] reaches out to aspect-oriented software architecture. Programming languages are still the most visible results of AOSD for the industry.*” [76].

The lack of support of aspects-oriented development at the early stages of development, such as requirements specifications and architecture description, has led to focused efforts to what is known as Early Aspects EA² (sometimes referred to as Aspects Oriented Requirements Engineering AORE). Simply, an *Early Aspect* is a crosscutting concern identified in early stages of development life cycle phases. An overview of different AORE approaches can be found in [13] and [118].

On the scenario-based specifications side, there is very little support for early aspects in scenarios. Whittle and Araujo [144] differentiate between an *aspectual* scenario and a normal scenario. Aspectual Scenarios in [144] are captured via so-called Interaction Pattern Specifications (IPs), whereas normal scenarios are specified in normal UML Sequence Diagrams. An IPS defines a pattern of interaction between its participants. To this extent, Whittle and Araujo regard aspects as patterns. This approach is similar to earlier works such as [53] and [34]. Kienzle [79] uses a technique similar to [144] in the sense that they regard an aspect as reusable functionality.

In our opinion, at the specification stage, *separation of concerns* should not be directly expressed in terms of reoccurring patterns in the specifications, as both approaches of [79] and [144] assume, even though this was successful in aspect-oriented programming (e.g., AspectJ), where reusable code

¹www.aosd.net

²www.early-aspects.net

parts (or reoccurring code patterns) are considered as aspects. The essence of *separation of concerns* in scenarios should take a more intuitive approach to identify *aspects*. Our approach to identifying *concerns* in specifications is based on identifying modes of the system. In a certain mode, the system is expected to execute a cohesive set of functions or tasks that together make up an *aspect*. These functions might reoccur in different modes, but they would behave differently in each. The mode defines a context for these functions and tasks, unlike existing approaches which assume these functions and tasks are themselves *aspects*.

To the best of our knowledge, our approach is the first to introduce mode-based *separation of concerns* in scenarios and availing contexts, in which scenarios are expected to run. We refer the reader to Chapter 4 for detail on modes and an overview of related work on separation of concerns. Our proposition is that if these contexts are structured, then scenarios themselves will be structured, and the problem of possible overlapping of the elicited scenarios is overcome by providing several corresponding viewpoints of those contexts.

5.2.4 Other Approaches to Realizing Inter-Scenario Relationships

So far we have discussed the most popular approaches and paradigms that directly addressed scenario composition, and that have received significant attention in the literature. There are, however, other works that realized the inter-scenarios relationships. The latter approaches, however, have not achieved significant results, though the ideas behind them are novel in most cases. In this section, We discuss examples of these approaches. We do not provide exhaustive listing of these approaches, primarily because they are marginal to the work presented here.

Valiache and Tanaka [142] realized that when building a state-machine of an object from its scenarios, different inter-scenario relationships result in different structures of the translated state machine. They proposed a *dependency graph* diagram, which uses notations similar to MSC, to relate scenarios together, in order to better define these relationships. They elaborated on this “dependency” relationships and classify them to relations, following earlier work of [valichie-relations], such as *timing*, *causal* and *generalization* relations [142].

In our opinion, enumerating the possible types of relationships between scenarios could be endless process, resulting in additional notations and they are likely to overlap in their semantics. Moreover, it will be impractical for designers to keep up with all these new notations and the design artefact itself is likely to be cluttered by the things that were supposed to help. The approach of Valiache and Tanaka is a sample of many other approaches that follows the UML’s stereotyping school of thought.

Other approaches resorted to a purely formal, and theoretical in most cases, approach to the

problem. Most prominently among those are the attempts to compose scenarios *algebraically*. Triggered Message Sequence Charts (TMSC) by Sengupta and Cleaveland [123] provide a rich set of algebraic composition such as: *parallel*, *delayed choice*, *sequential*, *recursive*, *conditional choice*, and logical AND operator. These operators are used to compose basic TMSCs to more compound scenario specifications. The TMSC approach also provides a full semantics framework to the MSC constructs, based on *algebraic process theory* [97, 119].

The original MSC standard has already provided algebraic semantics for MSC and HMSC constructs, and further work such as [97, 119] and [123] are definite improvements. However, a basic motivation of our work in this thesis is that scenarios should be kept as simple as possible to serve the purpose for which they were invented. So, even though the algebraic composition of scenarios is a useful approach, it does not serve the general application of scenarios, particularly when it comes to industry-level applications. The major concern is to reach reverse results and the final specifications get too complex, which discourages designers to maintain it.

5.2.5 Concluding Summary

In this section we surveyed major existing attempts to compose scenarios into more compound structures. We have given special attention to the popular standardized approaches, such as HMSC [72], IODs [103, 145], and we detailed their features. Moreover, we gave a comparative discussion of Aspect-Oriented development, in general, and Early Aspects [13, 118] in particular, where some approaches are concerned with separation-of-concerns in the initial specifications of scenarios.

We can safely conclude that the types of relationships assumed by existing approaches do not address the separation-of-concerns in scenarios, including those approaches falling under Early Aspects. A methodical approach for separation-of-concerns in scenario would minimize specifications partiality, a well known problem of scenario-based specifications in particular. Partiality always has been admitted by all approaches as an unavoidable problem, but no approaches provided a remedy for it.

In the rest of chapter, we propose a novel framework for specifying scenarios such that the resulting specifications are sets of scenarios structured within the system’s state-space, without inventing any new inter-scenario relationships. More concretely, we build on our formulation of *mode-based* specifications – presented in Chapter 4 – and use it as a framework to specify contexts (a context is represented as a *mode*) and scenarios which are iteratively specified within these contexts. The contexts (or modes) specifications allow us to encompass the boundaries of the state-space (defined by state variables) and uncover hidden “gaps” in the state-space that would possibly have been missed without such framework. Further scenarios are then developed to cover these gaps,

facilitating elaboration of requirements and minimizing the risk of partiality in the final specifications, and managing the complexity of the behavior space (i.e., the space of possible behaviors).

To put it simply, we use mode-based specifications to paint the big picture of the system landscape in a familiar (and abstract) automata-based model called *mode-machines* (Chapter 4), and we then specify the expected scenarios within the context created by *modes*. This big picture serves as a vehicle for communication between requirements analysts and domain experts, and creates a vocabulary to identify different scenarios. It is worth noting here that mode-based specifications allows for multiple descriptions of the state-space from different aspects, where each aspect is managed via a *mode-class* (see Chapter 4). This is a key distinguishing factor of applying modes as a design concept in software engineering.

5.3 Mode-Wise Structuring of Scenarios

We discussed in the previous section a representative subset of approaches and methods that attempted to compose scenarios, with various dialects, in some higher-level structures and composites that help to specify more compound systems. We identified the mainstream methods and we projected them onto two main techniques. The first assumes programmatic-like constructs to compose sequences within the same scenario, such as **loop**, **alt** and **par** operators. We can see obvious correspondences between these constructs and classic constructs used in general-purpose programming languages. The second structuring technique assumes a set of constructs that are similar to flowchart structures. See Section 5.2.5 for more details.

In this section, we propose a scenarios structuring technique that assumes a “lateral” direction compared to the current approaches. The approach works at the scenario-level so that it replaces the flowcharting composition technique, but it does not exclude the programmatic constructs within individual scenarios. Our approach can be thought of as a method to specify scenarios within automata-based frameworks. The fundamental concepts in this framework are the modes and mode-class concepts [7, 36, 107, 109, 113] that we detailed and elaborated in Chapter 4.

We first present a powerful synergy between the mode-based framework and scenarios. This synergy is the main motivation behind our proposal to structure scenario specifications, in addition to the motivation to overcome the problems suffered by existing approaches. Second, we present a procedure to employ our mode-based framework to structure scenarios, and we validate this approach by applying it to two well-known case studies in the computing literature – Steam Boiler Controller [2, 48] and the Mine Pump Controller ones

5.3.1 The Synergy Between Modes and Scenarios

The concepts of *mode and mode-class* [7, 36, 107, 109, 113] are classic concepts of modeling system behaviors. They originate from the paradigm of Hybrid Systems which involves both continuous and discrete behaviors. Chapter 4 provides a detailed discussion of the origins and motivations behind *modes*. A background discussion on Hybrid Systems can be found in [93, 139] .

Recalling from the discussion in Chapter 4, a *mode* is basically an abstract representation for a group of states, and can be characterized by a predicate invariant to all the states belonging to this mode. A mode-class is a collection of modes such that each state in the system must belong to one and only one mode. This means that the modes of a mode-class are: (1) non-overlapping, i.e., the states sets characterized by those modes are not intersecting, and (2) together they address a specific aspect of the system.

There could be several mode-classes of the same system, each of which describes the state-space from a certain aspect. Any state of the system must belong to single mode from each mode-class. So, in brief, mode-based specifications of the system comprise an arbitrary number of mode-classes, with the properties summarized above (see Chapter 4 for a detailed and formal definitions of these properties).

A mode-class is partial view of the system, however, this view is *aspectually complete*, i.e., it is *complete* as far as the *aspect* of this mode-class is concerned. The reader is referred to Chapter 4 for pictorial illustrations of these properties. So, we can safely say that a mode-class is capable of (aspectually) representing the system state-space and encompassing its boundaries.

On the scenario specifications side, a scenario is generally perceived as (and it actually is) an automaton fragment that is part of the overall system automaton. There is no doubt that scenario specifications suffer from the partiality problem (see Section 2.2 for elaborated definition of *partiality*) in addition to other drawbacks such the likelihood of lack of cohesion between scenarios. Scenario specifications considered as partial in two regards:

1. First, the weak sequential ordering [69, 71, 72] of messages means that, for a certain scenario, there is a possible arbitrary interleaving of other messages (absent from the current scenario) with the ones depicted on the scenario itself. This interleaving, when reflected on the automaton-based view of the scenario, means that there could be other states arbitrarily interleaved with those states directly translated from the scenario.
2. The other regard is that scenarios are not meant for (and practically they are incapable of) modeling end-to-end executions of the system. This may be feasible only in toy systems, or

specific short executions in real non-trivial systems. However, in general, a scenario is meant to be a short and “rounded” execution that is more likely to be a stage or phase of a larger long execution, too large to be depicted in a single scenario. So, typically, a scenario is part of end-to-end system execution.

The partiality of a scenario is not *aspectual* as in the case of a mode-class. A scenario specified freely without any context or scope is likely to have cross-aspects traversal in the system state-space.

If we compare the partiality drawbacks in scenario-based specifications to the aspectual and coverage features of mode-classes (coupled with disjointness of modes in the same class), we can identify a synergistic match between the two specifications types in the sense that the features of mode-based specifications powerfully augment those drawbacks of scenarios as follows:

1. The state-space coverage provided by mode-classes allows them to “encompass” the state-space boundaries and make sure that all states are included in the characterization provided by the mode-class specifications. In other words, each state in the state-space will be included in one of the modes of the mode-class at hand, and that each state will have the possibility to be “caught” in the scenario(s) that designers will specify in the corresponding mode.³
2. The multidimensional partitioning of the state-space, enabled by the possibility of several mode-classes (see Chapter 4), sets up a fertile environment for specifying scenarios. The importance of considering *aspects* in scenario specifications has already been motivated in other research work. Intuitively, allowing a scenario to focus on a certain aspect of the system greatly helps to elaborate on and compose more behaviors in that scenarios, and in turn this supports the elaboration and elicitation of the tasks that are crucial in the early stages of development. Mode-classes facilitate this disciplined multidimensional partitioning of the state-space where each mode-class can be considered as to partition the state-space from a unique aspect. Also, in Chapter 4, we have also discussed ideas and techniques that guide designers to specify the mode-class.
3. Furthermore, disjointness of modes (in the same mode-class) allows separation of distinct contexts in scenarios and, in turn, facilitates structuring a scenario that is too long and where contexts are mixed. The inter-context transitions are left to the transitions between modes (See definition of Mode-Machines in Chapter 4) instead of being blurred in the messages’ sequence.

³NOTE: In a mode, it is up to the scenario(s) flow to capture as much as possible of the states in this mode. The *mode* itself helps this capturing process by narrowing down the states from the large unmanageable number of possible states in the whole state-space to a relatively small number of those possible states that are characterized by the mode’s predicate. Please review the basic definitions in Chapter 4 for better clarification of the *state-space* and *modes* concepts.

This further narrows down the context in which the designer has to specify scenarios. In turn, this makes it easier for designers to capture and cover more states of the state group characterized by the mode.

Motivated by this synergy and the augmenting relation between mode-based and scenario-based specifications, we promote a procedure in the next section to structure scenarios using modes and mode-classes.

5.3.2 Scenarios Identification and Structuring

Given the motivation provided in this chapter for combining scenarios and modes, and given the foundations of mode-based specifications provided in Chapter 4, we are now in a position to test these ideas on some case studies to provide a proof of concept, which will serve to validate our assumptions. However, before we go ahead studying these ideas, we first need to define procedural steps to guide the system application of these ideas. This provides the methodical part of our approach. To this end, in this section we further elaborate a two-steps methodical technique to combine scenarios and mode-based specifications. We distinguish between the concepts of *Operational Behavior* and *Modal Behavior*. Our approach assumes a start-from-scratch writing of the system specifications and does not assume any pre-defined artefacts. In the next sections, we will apply this approach to a couple of real-world case studies. Below is an executive summary of this procedure:

Step 1: Identify Modal Behavior. To explain what *Modal Behavior(s)* means here we have to recall the definitions of mode-based specifications we put forward in Chapter 4. To put it informally: *for a system that operates in a set of modes; a mode is a set of conditions such that, when the system is in this mode, the system executes a cohesive set of functions that do not violate the mode's conditions, otherwise the system moves to another mode. The modes and their transition-relations constitute the Mode Machine of that system.* In this procedure, we consider the *Modal Behavior* to be defined as the Mode Machine of the system. If we have multiple Mode Machines – corresponding to different mode-classes – then they collectively represent the system's Modal Behavior, for these aspects that have been decided upon. The essential characteristic of *Modal Behaviors* is that they are independent behaviors. In other words, a *Modal Behavior* does not affect other Modal Behaviors, but it might affect some (or all) Operational Behaviors.

Step 2: Identify the Operational Behavior. First, let us understand what is meant by *Operational Behavior* here. Operational Behavior(s) is always driven by the system's functionality

requirements. For a certain function provided by the system, the Operational Behavior associated with this function may change from one mode to another while maintaining the same function. The *change* here does not necessarily mean that the system provides a different function, but rather the system should behave differently to maintain the same function in the new mode. However, a system could well change its function when it switches from one mode to another.

To illustrate the concept, consider the landing function of an airplane system; because the landing conditions vary (e.g., environmental conditions, fault conditions, auto-landing, etc.) then so the airplane system must be prepared for different scenarios (or behaviors) to land in various possible (combinations of these) conditions. As these conditions establish the Modal Behavior (as discussed in Step 1 above) the various landing scenarios, together, establish the Operational Behavior associated with the landing function.

Identifying the Operational Behavior assumes we have the Modal Behavior of this system already in place. Operational Behavior is identified and specified in light of the Modal Behavior. Recall that the Modal Behavior, as we defined in Step 1 above, involves a set of mode classes and their corresponding mode machines (see Chapter 4 for detail). For each mode in a mode-class, engineers shall elicit one or more Operational Behavior by asking “What would the system do while in this mode?” Operational Behavior is captured through simple scenarios. Each scenario is associated with one and only one mode. However a mode could have several scenarios elicited that operate or execute under the logical scope of this mode.

Step 2 shall be repeated for each mode in each mode-class. The result is a set of scenarios floating in the (structured) logical state-space of the system, represented by multiple mode-machines. Disjointness of modes prevents mixing of several concerns because modes define contexts for scenarios – this prevents specifying long, unmaintainable scenarios. To this extent, scenarios (i.e., Operational Behaviors) capture the intra-context in behavior of the system and Mode-Machines capture the inter-context one. The space-coverage feature of mode-classes (see Chapter 4) remedies the partiality of scenarios in the sense that (1) first, having (many) short and context-focused scenarios will result in less partiality of the overall scenario collection (as the focused scenarios will help capture more behaviors than longer scenarios where some behaviors may be overlooked), and (2) second, dissecting the state-space from different aspects (i.e., different mode-classes) and partitioning each aspect to several contexts (i.e., different modes) leaves little to no chance for hidden gaps in the state-space and assures higher specification coverage with scenarios and the location of missing scenarios.

The reader should note that mode-classes do not guarantee to *completely* eliminate partiality from the specifications. Partiality is primarily a result of the uncertainty in requirements collected

from users or system analysts. All specifications efforts (including our approach) are mere attempts to infer more behaviors and pose more questions to the requirements specifiers so that they may be able to elicit more requirements and behaviors. This technique presented above should be iterated until the specifications saturates.

In general, mode-classes would have a similar potential to structure other specifications types that suffer from problems such as those of scenarios. For examples, Goal-based specifications [11, 90] also suffer from partiality and lack of methodical way for specification.

5.4 Case Studies

Throughout this chapter we discussed and presented our approach for writing scenario-based specifications such that they are simple and structured. Our approach assumes the existence of mode-based specifications that guides the elicitation of scenarios and helps structure them in the state-space. We have provided, in Chapter 4, formal foundations and definitions of the concepts underlying mode-based specifications and we also provided a methodical technique for deriving modes and mode-classes given a set of system and state variables.

In addition to the foundation presented in Chapter 4, in this chapter we attempt to evaluate these propositions with two case studies. First, in Section 5.5 we apply our scenarios structuring approach to the Steam Boiler Controller (SBC) and we demonstrate the difference between long unmaintainable scenarios and short structured ones. The SBC case study already has modes set out in the original specifications but not formally specified. We provide a formal specifications for these modes and use them to structure scenarios. The second case study, the Mine Pump Controller (MPC), we go deeper and identify the MPC modes specifications in the first place, using the technique presented in Section 5.3.2, and then elicit the MPC scenarios accordingly. Finally we will summarize the case studies' findings and evaluations.

The case studies used in this chapter are popular in the literature, particularly in the related work publications. As we will see throughout this section, the results and findings of these case studies validate our assumptions and the propositions we put forward in our approach for scenarios structuring.

5.5 First case study: Steam Boiler Controller

The SBC case study is frequently used in relevant research work [2, 48] to demonstrate techniques for synthesis from declarative specifications such as *Goals* and *Properties*, as well as scenario-based specifications [2, 48]. In this section we first give a brief overview on the SBC requirements. Second we draw typical scenario of the SBC, without any structuring, to give the reader the intuition of a non-structured example of the specifications, and to help compare it with the results of our approach. Next we use our framework of modes and mode-classes, presented in Chapter 4, to iteratively introduce structuring into the specifications. We finally conclude with discussion and observations of the results.

Readers who are familiar with the steam boiler case study are invited skip to Section 5.5.2 where we start to apply our approach.

5.5.1 The Original Requirements of SBC

The Steam Boiler Controller is a software system that is supposed to maintain the water level within certain ranges, by controlling of a set of pumps, under different failure conditions. Figure 5.4 is a schematic illustration of the boiler vessel.

The system consists of the boiler vessel, instrumentation devices, an operator panel, and a controller SBC (our target system). Due to our interest here to demonstrate our concepts of structuring, we modified the requirements in [3] slightly to add more requirements in the requirement of the water-levels. We then analyzed the textual requirements and identified the different devices and signals necessary to clarify the system components for the reader here and to be able to set out the system variables used in later stages of the case study. The context diagram in Figure 5.5 illustrates the identified devices and their relationships .

The SBC textual requirements in [3] can be summarized in following points.

- The primary function of the SBC is as follows:
 - It controls the water pumps to maintain the water-level between the normal levels N_L and N_H , and stops the boiler should the water level risk reaching any of the boundary levels M_L and M_H .
 - It detects various system failures, via the signals generated by the monitoring devices. It reports these failures and handshakes their acknowledgments with the Operator.
 - It applies different water-level control strategies according to current mode.
- The data transfer is managed as follows:

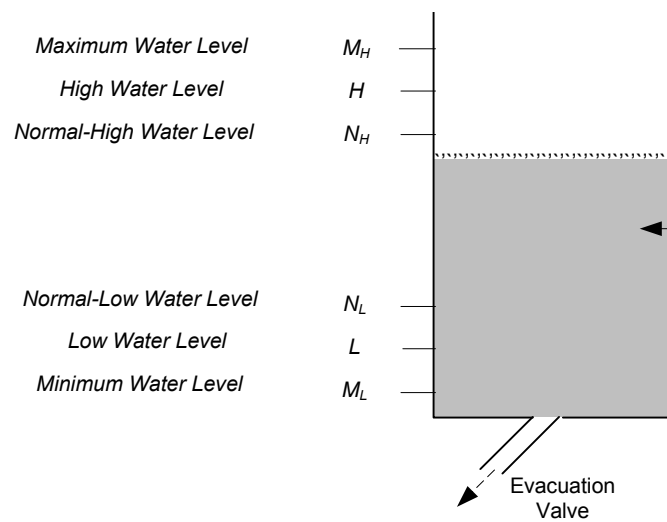


Figure 5.4: Schematic diagram of the Steam Boiler system

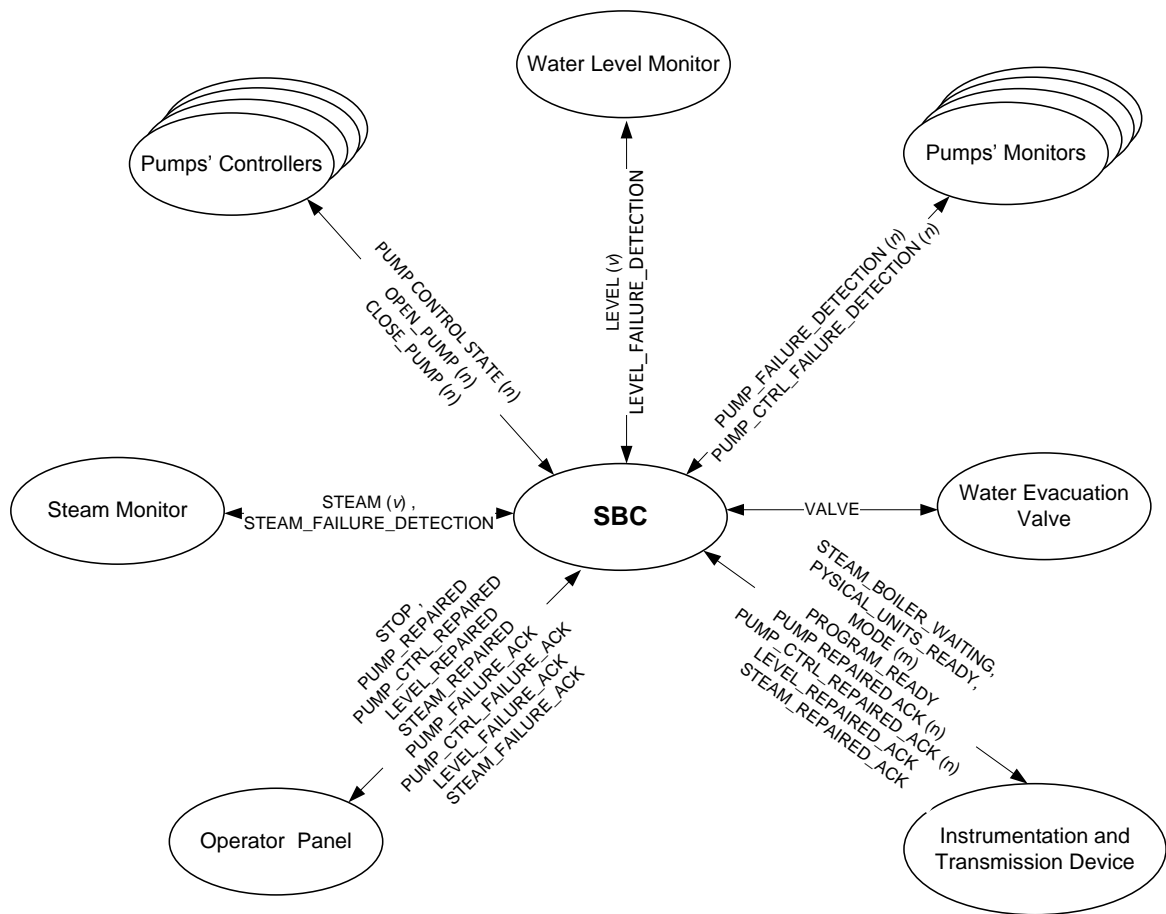


Figure 5.5: Context diagram of the Steam Boiler system

- All communication between the controller and the boiler vessel’s devices occurs in discrete rounds, once every 5 seconds. In each round, all units send information messages to the controller, and the controller responds by sending messages to the units. All communication is assumed to take place instantaneously.
- These messages are received as a vector of values called the *input transmissions*, with frequency of one transmission every 5s.
- The SBC continuously sends control signals to the various units in the *output transmission* vectors.
- Some signals should be sent only once (e.g., the control signal of opening or closing a pump), and others must be sent in each transmission (e.g., the pump state signal). An absent signal will have a value of nil in its corresponding position in the transmission vector.
- The functions of each device, as shown in Figure 5.5, is as follows:
 - Water-volume device: it has a sensor to measure the water volume in the tank, and another sensor to indicate the measurement device itself is OK.
 - Four pump-control devices, each of which does the following:
 - * Accepts from the SBC a signal to open/close the pump and controls the pump accordingly.
 - * Reports to the SBC the pump’s operational status (i.e., Open or Closed) in each transmission cycle.
 - * Reports to the SBC whether the pump is defective or not, in each transmission cycle.
 - Four pump-monitoring devices, each of which does the following:
 - * Reports to the SBC, in each transmission cycle, if the water is flowing from the pump to the boiler or not.
 - * Reports to the SBC, in each transmission cycle, if the water-flow monitoring sensor is defective or not.
 - Steam Exhaust monitoring device that provide the following:
 - * Reports to the SBC, in each transmission cycle, the steam evacuation rate, in each transmission cycle.
 - * Reports to the SBC, in each transmission cycle, if the steam-monitor is defective or not, in each transmission cycle.

- The SBC operates in five *modes*:
 1. **Initialization mode:** the SBC waits for the steam boiler to signal its readiness for operation. Once the controller receives a signal that the boiler is ready, it then tests the amount of steam escaping from the boiler. If this is nonzero, it enters the emergency stop mode. Otherwise, it either drains the water level to or activates a pump to raise the water level to N_L . Once the range of normal water levels has been reached, the controller sends a signal to the physical units, waits for acknowledgments, and then proceeds to the next mode according to the health-status of the physical units.
 2. **Normal mode:** all the physical devices are working properly and the water-level is under control. The SBC makes its decisions to turn pumps on or off based on the current water level. No action is taken if the water level lies in the range $[N_L, N_H]$. The SBC performs transitions to other modes based on the health-status of the physical units and the rate at which steam is being emitted.
 3. **Degraded mode:** this mode is activated when one or more physical units has failed, but the water-level measuring unit is OK. The SBC has to apply some control strategy that makes use of the operational physical units to keep the water-level within limits.
 4. **Rescue mode:** this mode is activated if the water-level measuring unit is failed, and all other units are assumed to be healthy. The SBC has to estimate the water-level value using some calculations.
 5. **Emergency-stop mode:** This mode is reached if the water-level risks reaching its maximum limits, or in the case of physical unit failure that is beyond the *DEGRADED* or the *RESCUE* modes, or in case of transmission error. The SBC has to stop the boiler in this mode.

Full details of the requirements can be found in [3].

5.5.2 Naive Scenario Specifications of SBC

In the previous section we gave a brief description of the steam boiler's textual specifications as supplied in the literature. Before we go ahead and applying our scenarios structuring approach, we think it useful to give the reader intuition of how these scenario specifications would look like without structuring. This, on one hand, shows the complexity and irregularity (see Chapter 4) of the steam boiler system itself, and, on the other hand, it helps to show case the benefit of introducing structure in scenarios (as per our approach).

Table 5.1: Messages sent by the SBC

Messages issued by the : SBC	Description
MODE (m)	signal to the operator panel The current mode assumed by the SBC
PROGRAM_READY	Sent during initialization to tell the boiler vessel that the program is ready to start operation
OPEN_PUMP (i)	signal to switch on the i th pump
VALVE (n)	signal to open the water evacuation valve
CLOSE_PUMP (n)	signal to switch off the i th pump.
PUMP_FAILURE_DETECTION (n)	signal to inform the Operator Panel to inform of failure detected in operation of the i th pump
PUMP_CTRL_FAILURE_DETECTION (n)	signal to inform the Operator Panel to inform of a failure in the device controlling the i th pump
LEVEL_FAILURE_DETECTION	signal to inform the Operator Panel to inform of failure detected in the water-level measuring unit
STEAM_FAILURE_DETECTION	signal to inform the Operator Panel to inform of failure detected in the steam flow measuring unit
PUMP_REPAIRED_ACK (n)	signal to the Operator Panel to acknowledge the repair of the i th pump
PUMP_CTRL_REPAIRED_ACK (n)	signal to the Operator Panel to acknowledge the repair of the device controlling the i th pump
LEVEL_REPAIRED_ACK	signal to the Operator Panel to acknowledge the repair of the water-level measuring unit
STEAM_REPAIRED_ACK	signal to the Operator Panel to acknowledge the repair of the steam flow measuring unit

Table 5.1 and Table 5.2 below list the messages to be exchanged in the scenarios between the SBC and the entities in the system. The table briefly describes each message as a quick reference for our reader.

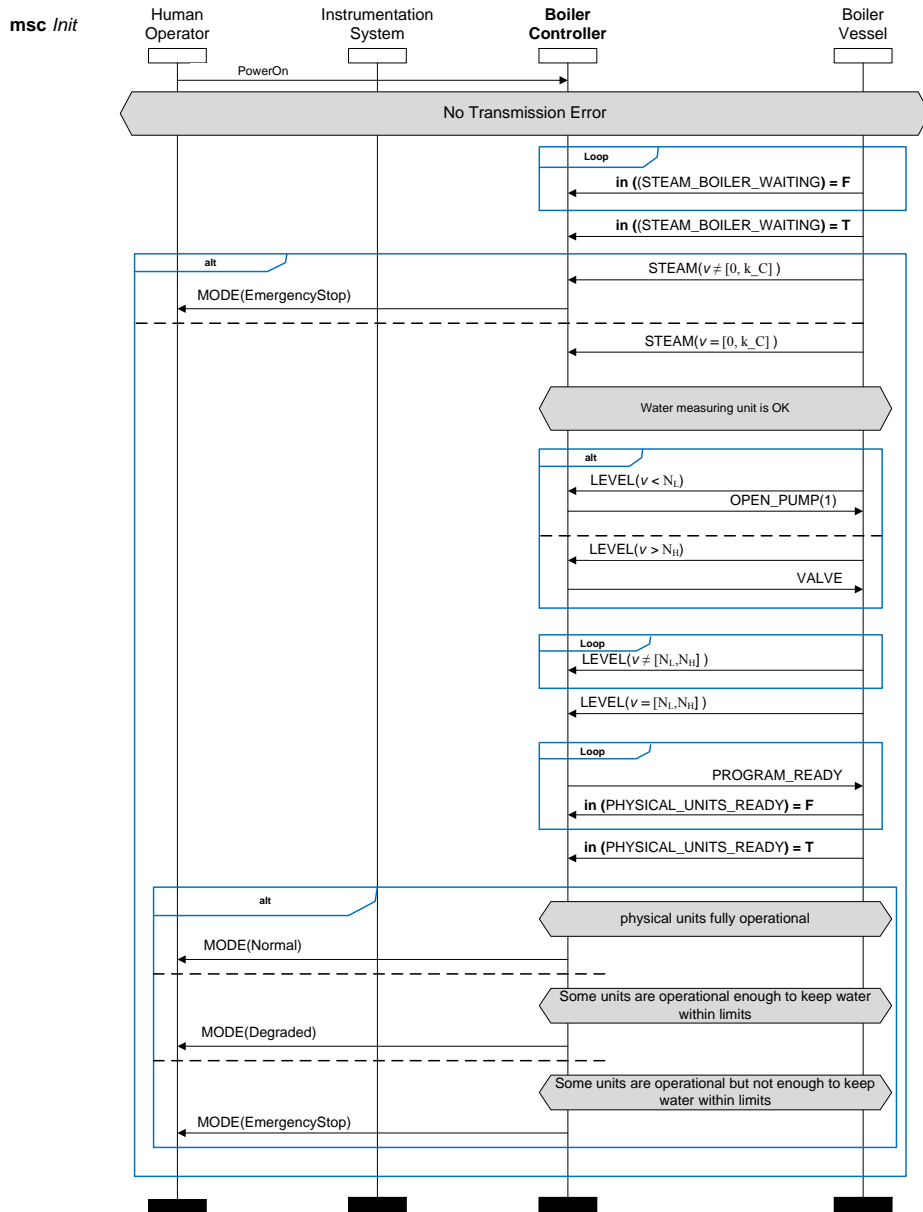
A naive rendering of the SBC's behavior scenarios (i.e., directly extracted from the textual specifications) would lead initially to five scenarios corresponding to the five modes. In Figures 5.6 and 5.7 we show two of these scenarios; the **msc *Init*** scenario describing the initialization phase of the SBC, and the **msc *Norm*** scenario describing the SBC behavior during normal operation.

From scenarios depicted in Figures 5.6 and 5.7, we can notice the following

- Although these scenarios are not short, neither of them could accommodate much of the relevant exceptions. For example, in the **sbc-init** scenario, the scenario flow assumed the water-level and steam-flow measuring units are OK in order to keep on and show the operation of switching to the various modes. We have two ways:

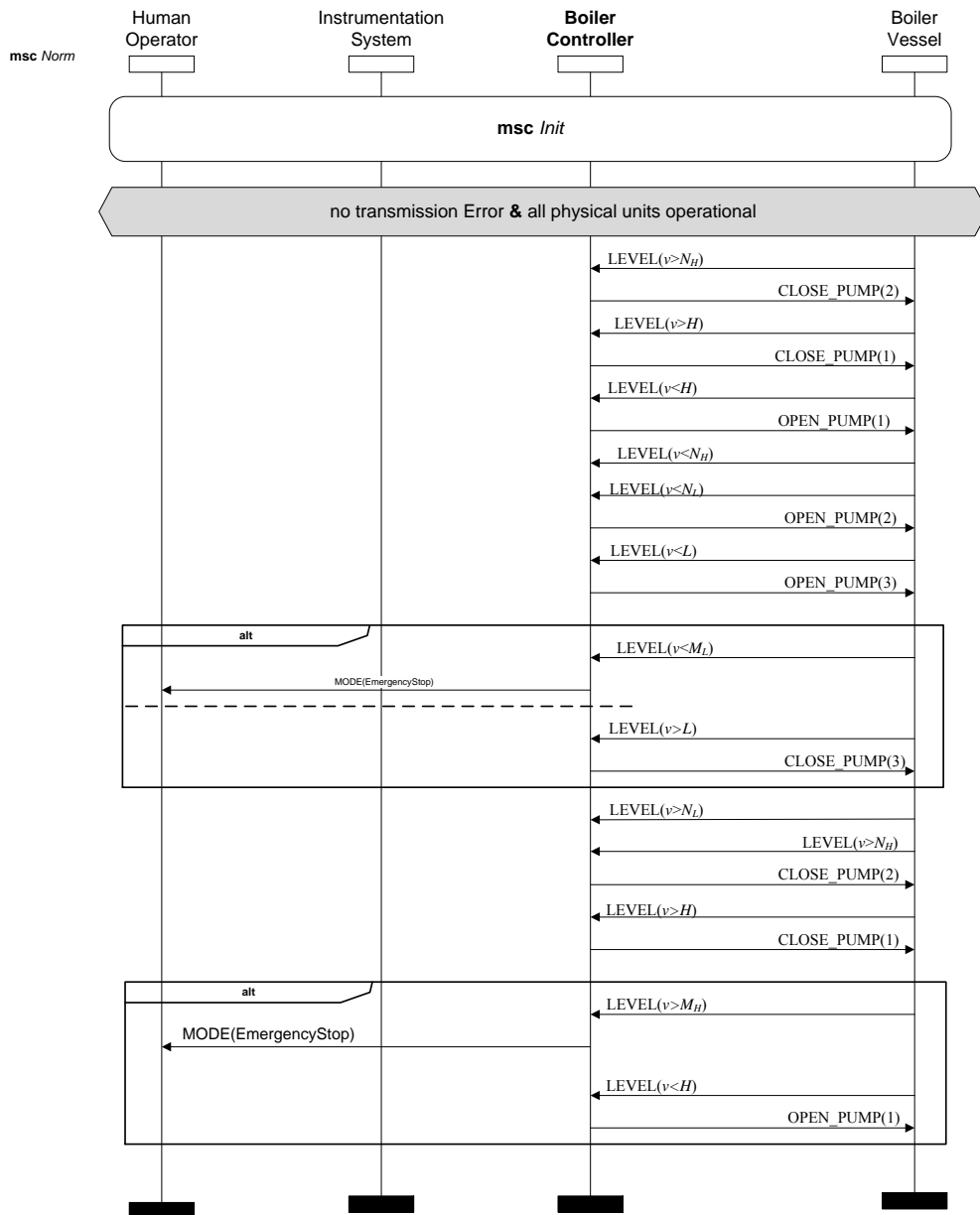
Table 5.2: Messages received by the SBC

Messages received by the SBC	Description
STOP	signal to inform the SBC to stop its operation
STEAM_BOILER_WAITING	signal to inform the SBC that the boiler vessel is ready, in response to the message <i>PROGRAM_READY</i>
PYSICAL_UNITS_READY	signal to inform the SBC that the physical units are ready for operation
PUMP_CTRL_STATE (<i>i,b</i>)	signal to inform the SBC of the current ON/OFF state of the <i>i</i> th pump
LEVEL (<i>v</i>)	signal to inform the SBC the current level of water in the boiler tank
STEAM (<i>v</i>)	signal to inform the SBC the current rate of the steam flow out of the boiler tank
PUMP_REPAIRED (<i>n</i>)	signal to inform the SBC that the <i>i</i> th pump is now repaired
PUMP_CTRL_REPAIRED (<i>n</i>)	signal to inform the SBC that the <i>i</i> th pump's control devices is now repaired
LEVEL_REPAIRED	signal to inform the SBC that the water-level measuring unit is now repaired
STEAM_REPAIRED	signal to inform the SBC that the steam- flow measuring unit is now repaired
PUMP_FAILURE_ACK (<i>n</i>)	signal from Operator Panel to acknowledge the <i>i</i> th pump failure that has been detected by the SBC
PUMP_CTRL_FAILURE_ACK (<i>n</i>)	signal from Operator Panel to acknowledge the <i>i</i> th pump's control device failure that has been detected by the SBC
LEVEL_FAILURE_ACK	signal from Operator Panel to acknowledge water-level measuring unit failure that has been detected by the SBC
STEAM_FAILURE_ACK	signal from Operator Panel to acknowledge steam-flow measuring unit failure that has been detected by the SBC



- 1- It is difficult in this diagram to express the possible change of commsOK at any point.
- 2- To make the depiction easier, we assumed that the water level will not fall during the alternative of checking the water level.
- 3- Also local actions are not clear here, e.g., the action of modifying the new failure mode info in the internal state of the SBC component. This complicates the synthesis process
- 4- Identifying the state variables will help also to express if all, some or none of the physical units are ready or not. This clarifies the ambiguity in the given requirement that says "the program enters either the mode normal if all the physical units operate correctly or the mode degraded if any physical unit is defective." which contradicts and does not distinguish between the conditions of the modes degraded and rescue

Figure 5.6: SBC initialization scenario (unstructured)



1- we made a big assumption here is that we do assume the water level can not suddenly jump directly to M_H or M_L skipping other levels. Otherwise, we would add **alt** construct at each step.

2- The first **alt** construct is inaccurate. The problem is the depicted sequence assumes that the scenario flow rejoins after this alternative. This is incorrect as the requirements says that if we switch to EmergencyStop, then there is no further flow. The only workaround here is to split this scenario to multiples ones whenever we face such situation...leading to many scenarios (each has to have a similar initialization flow).

Figure 5.7: SBC behavior in Normal mode

- If we tried to consider all those cases, the scenario would have exploded in length and complexity of nested constructs.
- Alternatively, we may think of capturing those cases in additional scenarios, however we will still have to write (in each additional scenario) the initial necessary steps that lead to occurrence of the target case or exception, and these steps will be similar in (almost) all scenarios. This alternative results in too many redundancies between many scenarios such that two scenarios may be exactly the same except for one or two steps. Designers then will have to take the tricky decision to either go for a very long scenario with complex nesting of conditions, or opt for several scenarios with unmaintainable redundancies.
- Another issue is that scenarios are commonly assumed to start with any possible step (i.e., state or event) in the system execution. Requirements specifiers do not know (and do not have the discipline to help decide) when a scenario starts and when it should be ended. Typically, this is driven by 'stories' told by customers. Approaching scenario specifications in this way might bury other important scenarios within one of these stories, in addition to leading to the issues discussed in the previous point.

In the following section we introduce structure to these scenarios in an attempt to overcome the aforementioned issues. More specifically, we will use our framework, proposed in Chapter 4, to refactor the SBC scenarios (including the ones in Figure 5.6).

5.5.3 Structuring the SBC Scenarios

In Section 5.3.2 we provided a methodical procedure for using the mode-based framework, that we proposed in Chapter 4, to specify structured scenario specifications. It's worth mentioning that the framework can be used in the beginning of the specification task, where no scenarios have yet been drawn, and also can be applied to pre-existing (e.g., legacy) scenarios. In this section we apply our structuring framework to the steam boiler case study to specify the SBC behavior, assuming no scenarios have been drawn yet. It will then be intuitive enough to think of how we can apply it to pre-existing SBC scenarios such as those in Figure 5.6.

We start the procedure, as per the steps given in Section 5.3.2, by identifying the system variables that define the state-space of the SBC. Next, we iteratively try to identify the mode-classes of the SBC and draw the relevant scenario inside each mode.

We chose the SBC as the first case study to show our structuring approach for of several reasons. First, the original textual specifications have prescribed a set of modes and described the behavior in

each mode. This helps in our demonstration as it gives the reader a sense of how *modes* are handy constructs that can be used by system analysts in the first place. Second, having explicit modes specifications in the initial specifications helps us in the SBC case study to focus on drawing scenarios (aspectually) within modes. Finally, as we will see later in this case study, we will show how the pre-specified modes are not enough to capture all the aspects of the SBC behavior, and we will propose an additional mode-class to further simplify scenarios. The step of adding one more mode-class, itself, will help us to examine the impact of Requirements Change on our approach, and we will place particular emphasis on this point within the discussion and the conclusion provided at the end of the case study in Section 5.5.3.3.

5.5.3.1 Identifying domain variables

As we discussed earlier in Chapter 4, the decisions involved in selecting the domain variables are highly dependent upon the application domain, the target platform and the designer's intuition. This can easily be appreciated from the example of modeling the status of, say, a 32-bit counter. Designers may choose to model the counter's status as two 16-bit variables, or four nibbles, or even 32 Boolean variables.

We made a more realistic analogy in Chapter 4 (section 4.4) between this process and the process followed in Object Oriented Analysis and Design OOAD [80, 103, 116] methods and techniques to identify objects in the system. While the (OOAD) method aims, primarily, to define a structural model of the system, one can think of the system state variables identification and selection as defining the state-space (and integrally contribute to defining the behavior space) of the system.

We analyzed the requirements in [3] (outlined in Section 5.5.1), and we identified the set of state variables specified in Table 5.3.

As discussed in Chapter 4, identifying the system variables is an intuitive effort in the first place, and should be supported by domain expert's knowledge. Nevertheless, some disciplined and methodical techniques already exist in the literature to guide this effort. In our case studies here, we follow [141] the general guidelines of the *4-Variable* method to identify the system variables. The 4-Variables Model assumes 4 types of variables: *input*, *monitored*, *controlled* and *output*. To focus on the case studies' objectives, we assumed the *monitored* variables are the same as *input* variables, and analogously, we assumed the *controlled* are the same as *output* variables.

In addition to the input and output sets of variables, there are *internal* state variables (e.g., timers, counters, etc.) that are not necessarily input or output but are modeled by designers to

Table 5.3: The identified state-variables of the Steam Boiler Controller

Predicate Variable	Range Values	Function Specification
s_Stop	T, F	$STOP \Rightarrow s_Stop$
$s_CommsErr^*$	T, F	T : for any error in the physical transmission line(s) between the boiler and the SBC F : if no transmission errors.
$s_PmpState[i]$	T, F	$PUMP_CONTROL_STATE(i, ON) \Rightarrow s_PmpState$ $PUMP_CONTROL_STATE(i, OFF) \Rightarrow \neg s_PmpState$
$s_PmpErr[i]^*$	T, F	set to T if an erroneous behavior detected in the i th pump's dynamics
$s_PmpCtrlErr[i]^*$	T, F	set to T if an erroneous behavior detected in the pump controller's physical dynamics
$s_WaterUnitErr^*$	T, F	set to T if an erroneous behavior detected in the water-level measuring unit
$s_SteamUnitErr^*$	T, F	set to T if an erroneous behavior detected in the steam-flow measuring unit
$s_PmpOK[i]$	T, F	$(PUMP_CONTROL_STATE(i, CIRC) \wedge \neg s_PmpErr \wedge \neg s_PmpCtrlErr) \Rightarrow PmpOK[i]$
$s_SteamFlow$	$[0, W]$	$(STEAM(v) \in [0, W]) \Rightarrow (s_SteamFlow = STEAM(v))$
$s_SteamOK$	T, F	$\neg s_SteamUnitErr \Rightarrow s_SteamOK$
$s_EmergencyWL$	T, F	$(LEVEL(v) \leq M_L) \vee LEVEL(v) \geq M_H \Rightarrow s_EmergencyWL$ $s_EmergencyWL \Rightarrow \neg(s_VeryLowWL \vee s_LowWL \vee s_NormalWL \vee s_HighWL \vee s_VeryHighWL)$
$s_VeryLowWL$	T, F	$LEVEL(v) \in [M_L, L] \Rightarrow s_VeryLowWL$ $s_VeryLowWL \Rightarrow \neg(s_EmergencyWL \vee s_LowWL \vee s_NormalWL \vee s_HighWL \vee s_VeryHighWL)$
s_LowWL	T, F	$LEVEL(v) \in [L, N_L] \Rightarrow s_LowWL$ $\Rightarrow \neg(s_VeryLowWL \vee s_EmergencyWL \vee s_NormalWL \vee s_HighWL \vee s_VeryHighWL)$
$s_NormalWL$	T, F	$LEVEL(v) \in [N_L, N_H] \Rightarrow s_NormalWL$ $s_NormalWL \Rightarrow \neg(s_VeryLowWL \vee s_LowWL \vee s_EmergencyWL \vee s_HighWL \vee s_VeryHighWL)$
s_HighWL	T, F	$LEVEL(v) \in [N_H, H] \Rightarrow s_HighWL$ $s_HighWL \Rightarrow \neg(s_VeryLowWL \vee s_LowWL \vee s_NormalWL \vee s_EmergencyWL \vee s_VeryHighWL)$
$s_VeryHighWL$	T, F	$LEVEL(v) \in [H, M_H] \Rightarrow s_VeryHighWL$ $s_VeryHighWL \Rightarrow \neg(s_VeryLowWL \vee s_LowWL \vee s_NormalWL \vee s_EmergencyWL)$
$s_FailureMode$	(Init, Norm, Degr, Rescue, Stop)	$M_{inti} \Rightarrow (s_FailureMode = Init)$ $M_{norm} \Rightarrow (s_FailureMode = Norm)$ $M_{degrd} \Rightarrow (s_FailureMode = Degr)$ $M_{rescue} \Rightarrow (s_FailureMode = Rescue)$ $M_{stop} \Rightarrow (s_FailureMode = Stop)$ (The functions of the predicates $M_{inti}, M_{norm}, M_{degrd}, M_{rescue},$ and M_{stop} are specified in Section 5.5)

capture some internal state of the system. The input, output and internal variables together determine the state of the system at any point of time [54]. The set of the system's state variables is represented by the *union* of the sets of input and output internal variables (Table 5.3). In case of replicated variables (e.g., all the state variables associated with *water pumps*.) we use an array-like variable to identify them. We use the prefix '*s_*' to refer to any of these state variables. Table 5.3 lists the state variables we identified for the SBC system.

5.5.3.2 Specifying mode-classes and Scenarios: iteration #1

Given that we identified the SBC's state variables (Table 5.3), we then shall use these variables to compose formal specifications (in simple predicate logic) for SBC's mode-classes. The initial textual specifications of SBC mentions explicitly a number of modes of the SBC. We deliberately present the SBC as the first case study here because it already predefines a set of modes, and this helps in incremental demonstration of our structuring approach. We will compose the formal specifications for these modes and use it as a seed of our modes specifications.

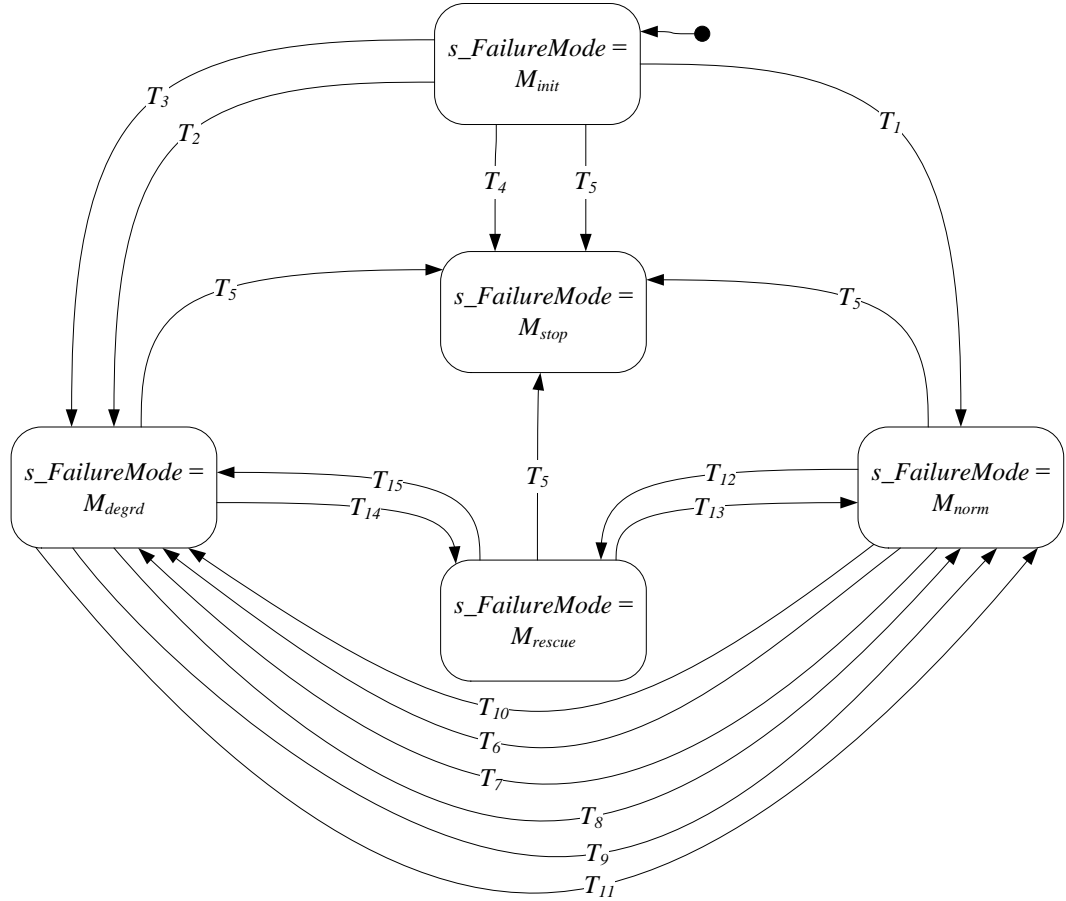
Let us refer to the mode-class *Failures Modes*⁴ as MC_{fm} and consider it as the initial part of our structured specifications. We then specify the corresponding mode-machine MM_{fm} by extracting – from the textual requirements – the predicates that characterize the modes in MC_{fm} . Recall that a mode is characterized by a predicate as explained in Chapter 4. We also specify the transition relations between those modes, according to the textual specifications. The transition diagram of the MM_{fm} is shown Figure 5.8.

We use the variables in Table 5.3 to compose the modes' predicates. We stick to the basic predicate logic to codify the mathematical functions of the these predicates. The granularity of the state variables facilitates the composition of the these predicates. However, when it is necessary to specify more composite predicates, and none of the variables defined (so far) is suitable, we will define new variables to name the mathematical functions of those predicates. For example, we used the variable $s_FailureMode$ to refer to the current failure mode from the MC_{fm} class. The specifications of the predicates functions of all failure modes of MM_{fm} are detailed in Table 5.4.

⁴Note: we call these modes as *failure modes* because they are so called in the given initial specifications.

Table 5.4: Predicate functions of the transitions in the mode machine MM_{fm}

$M_{init} \Rightarrow (s_SteamFlow = 0) \wedge$ $\neg(s_SteamUnitErr \vee s_EmergencyWL \vee s_WaterUnitErr) \vee$ $\neg s_CommsErr$
$M_{norm} \Rightarrow \neg(s_SteamUnitErr \vee s_EmergencyWL \vee s_WaterUnitErr) \wedge$ $\neg(\forall i : s_PmpErr[i] \vee s_PmpCtrlErr[i]) \vee$ $\neg s_CommsErr$
$M_{degrd} \Rightarrow ((s_SteamUnitErr \vee s_EmergencyWL \vee s_WaterUnitErr) \wedge$ $\neg(\exists i : s_PmpErr[i] \vee \exists i : s_PmpCtrlErr[i])) \vee$ $\neg s_CommsErr$
$M_{rescue} \Rightarrow s_WaterUnitErr \wedge$ $\neg(s_SteamUnitErr \vee s_EmergencyWL) \wedge$ $\neg(\forall i : s_PmpErr[i] \vee s_PmpCtrlErr[i]) \vee$ $\neg s_CommsErr$
$M_{stop} \Rightarrow s_Stop \vee s_EmergencyWL \vee s_CommsErr \vee$ $(s_SteamUnitErr \wedge s_WaterUnitErr) \vee$ $(\forall i : s_PmpErr[i] \vee s_PmpCtrlErr[i]) \vee$


Figure 5.8: Failure Modes Machine MM_{fm}

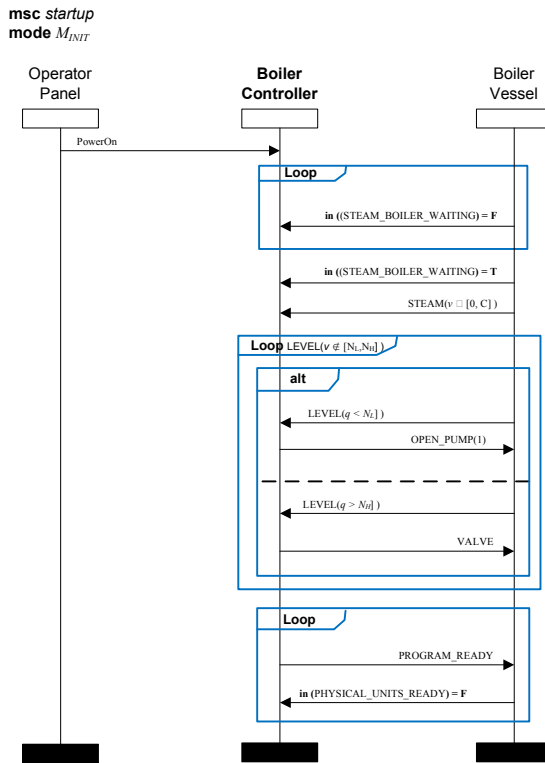


Figure 5.9: *Init-1* and *Norm-1* scenarios specified under the scope of the modes *Initialization* and *Normal*, respectively. These scenarios are shorter than their unstructured version in Figures 5.6 and 5.7 specified directly from initial specifications.

The specification of the transition relationship between the SBC is as follows:

Having defined this initial modal part of the SBC behavior, we then turn to the scenario behavior. The challenge is to compose the scenarios that fit within the scope of each those modes. According to the procedure in Section 5.3.2, a scenario scoped by a mode M can not include any interaction (actions or messages) that causes a transition outside the scope of M . More formally, a scenario scoped by a mode M can not include any interaction that results in a system state that does not satisfy the predicate P_M . The two scenarios in Figure 5.9, *mssc Init-1* and *mssc norm-1* are specified under the scope of the modes *Initialization* and *Normal*, respectively. Comparing the scenarios *mssc Init-1* and *mssc norm-1*, in Figure 5.9, to their corresponding versions *mssc Init* and *mssc Norm* in, respectively, Figures 5.6 and 5.7, we can observe the following:

1. **Shorter scenarios.** The scenarios *Init-1* and *norm-1* are much shorter. The interactions

Table 5.5: Predicate functions of the transitions in the mode machine MM_{fm}

$\mathbf{T}_1 \Rightarrow @(\text{PHYSICAL_UNITS_READY})$ WHEN($s_FailureMode=Init$) /($s_Failure=Norm$)
$\mathbf{T}_2 \Rightarrow @(\text{PHYSICAL_UNITS_READY})$ WHEN($s_FailureMode=Init$) /($s_FailureMode=Degr$, PUMP_FAILURE_DETECTION(i))
$\mathbf{T}_3 \Rightarrow @(\text{PHYSICAL_UNITS_READY})$ WHEN($s_FailureMode=Init$) /($s_FailureMode=Degr$, PUMP_CTRL_FAILURE_DETECTION(i))
$\mathbf{T}_4 \Rightarrow @(\text{STEAM_BOILER_WAITING})$ WHEN($s_FailureMode=Init$) / $s_FailureMode=Stop$
$\mathbf{T}_5 \Rightarrow @(s_CommsErr \vee STOP)$ WHEN(T) / $s_FailureMode=Stop$
$\mathbf{T}_6 \Rightarrow @(\exists i : s_PmpErr[i])$ WHEN($s_FailureMode=Norm$) / $s_FailureMode = Degr$, PUMP_FAILURE_DETECTION(i)
$\mathbf{T}_7 \Rightarrow @(\exists i : s_PmpCtrlErr[i])$ WHEN($s_FailureMode=Norm$) / $s_Failure = Degr$, PUMP_CTRL_FAILURE_DETECTION(i)
$\mathbf{T}_8 \Rightarrow @(\text{PUMP_REPAIED}(i))$ WHEN($s_FailureMode=Degr$) / $s_FailureMode = Norm$, PUMP_REPAIRED_ACK(i)
$\mathbf{T}_{10} \Rightarrow @(s_SteamUnitErr)$ WHEN($s_FailureMode=Norm$) / $s_FailureMode = Degr$, STEAM_FAILURE_DETECTION
$\mathbf{T}_{11} \Rightarrow @(\text{STEAM_REPAIED})$ WHEN($s_FailureMode=Degr$) / $s_FailureMode = Norm$, $s_SteamUnitErr=F$, STEAM_STEAM_REPAIRED_ACK
$\mathbf{T}_{12} \Rightarrow @(s_WaterUnitErr)$ WHEN($s_FailureMode=Norm$) / $s_FailureMode = Rescue$, LEVEL_FAILURE_DETECTION
$\mathbf{T}_{13} \Rightarrow @(\text{LEVEL_REPAIED})$ WHEN($s_FailureMode = Rescue \wedge (\forall i : \neg s_PmpErr[i] \wedge \forall i : \neg s_PmpCtrlErr[i])$) / $s_FailureMode = Norm$, $s_WaterUnitErr = F$, LEVEL_REPAIRED_ACK
$\mathbf{T}_{14} \Rightarrow @(s_WaterUnitErr)$ WHEN($s_FailureMode = Degr$) / $s_FailureMode = Rescue$, LEVEL_FAILURE_DETECTION
$\mathbf{T}_{15} \Rightarrow @(\text{LEVEL_REPAIED})$ WHEN($s_FailureMode = Rescue \wedge (\exists i : s_PmpErr[i] \vee \exists i : s_PmpCtrlErr[i])$) / $s_FailureMode = Degr$, $s_WaterUnitErr = F$, LEVEL_REPAIRED_ACK

concerned with modes-transitions specified in the scenarios in Figure 5.6 no longer exist in the structured scenarios in Figure 5.9. Those interactions are now in the mode machine MM_{fm} . Moreover, the *conditions* expressions in Figure 5.6 attempt to specify (unfortunately in an *ad hoc* way) a context for the scenarios. These expressions are no longer needed in the structured scenarios, because these scenarios have had their contexts setup (rather in a disciplined way) by the mode machine MM_{fm} . Note that designers may still need to specify conditions within scenarios, but for other purposes (for e.g., for model synthesis purposes, as we do in Chapter 6).

2. **Extracting the 'exceptions' logic.** The transitions that were lengthening the scenarios in Figures 5.6 and 5.7 are now modeled as mode transitions in MM_{fm} .
3. **Scoped and focused scenarios.** As a consequence of point 2, the task of scenarios elicitation is reduced to describing those scenarios that do not involve exception cases, such as physical units failures. In other words, the decision-space of *what and/or which way the scenario flow shall go?* is much reduced.
4. **Fewer scenarios.** Without structuring: at some point in the scenario's flow we might need to branch to different alternative paths. In MSC and its variants (See Chapter 2), this is typically expressed by the **alt** or **opt** constructs. As the concepts underlying these constructs are drawn from general programming languages, they do not scale well with system size, particularly at the architectural level. Using these constructs might be useful for composing the alternative branches sequentially, in cases where, for example, the branches' flows will rejoin shortly and are expressible in the same scenario, without complicating it. However, in other not uncommon cases, where the branches are themselves too long to fit together in the same scenario (or even not rejoining at all, e.g., leading to acceptance states) resulting in a scenario too long to be comprehended, these programmatic constructs are no longer useful. The common workaround is to split the scenario into many new scenarios as the number of long branches, resulting in larger number of scenarios. With structuring; our approach avoids this by factoring out those cross-modes steps, leaving the scenario to its original job at which it excels.
5. **Scenarios are easier to write, read and maintain.** This is a consequence of points 2 and 4 mentioned above.

Although we criticize the flowchart-like features that are pervasive in various dialects of scenario-based specifications (see Chapter 2 for detail) because they could easily be overused and distort the basic usage of the scenarios format, we still see these features as useful in avoiding

repetitive steps in the scenario's flow – as long as these features are not overused. This can be illustrated in the (structured) scenarios of Figure 5.9, where the two scenarios are reasonably short and more importantly are focused on the contexts defined by MC_{Init} and MC_{Norm} . For example, in Figure 5.9 the loop construct in both scenarios proves useful to compact, in a single scenario all the possible scenarios that would be needed to express the behavior in the respective modes. Without loops we would need:

- A set of scenarios to specify the SBC behavior in MC_{Norm} to control the water-level as water increases and decreases,
- A similar set of scenarios to specify the behavior at 'reversing' points (i.e., the point at which the level reverses from increase to decrease),
- And repeat all of these scenarios to enumerate water-level points over the range $[M_L, M_H]$.

In this Iteration of the SBC case study we demonstrated how modes can introduce structure into scenarios. So far, however, we have yet not examined how designers could specify mode-classes. Recalling that we deliberately started with this case study because it has the mode-based specifications already provided in the input textual requirements, and this allowed us in this iteration to focus on the specification of scenarios in (pre-existing) modes. In the next iteration of this case study we will attempt to identify another mode-class, $MC_{w\ell}$, that will further simplify the scenarios. We will notice an important consequence of overuse the mode-classes; designers may continue to stretch the specifications in modes directions that might be counter productive in terms of the complexity of the final specifications.

5.5.3.3 Conclusion of the SBC case study

The SBC case study is a real world system that combines both the regular behaviors of hybrid systems [93, 139], which are characterized by the regularity of the physical process of water-level changes, and also, the irregular computing behaviors which have been imposed by the Failure Modes of the SBC, and the various transitions according to arbitrary conditions prescribed as constraints by the initial requirements.

We attempted in our study of the SBC to demonstrate the idea of simplifying and structuring the SBC's scenario-based specifications by partitioning the SBC's state-space into *modes* that are grouped together in *mode-classes* (See Chapter 4 for a detailed discussion of *modes* and *mode-classes*). The original SBC textual requirements specify, informally, a set of Failure Modes of the system. The

pre-specified modes should have given the reader some sense of how the concept of modes is already employed (but informally) in requirements specifications particularly in hybrid systems or control-oriented software applications. Also we attempted in this section to extend the concept to scenario-based specifications based on the sound foundation we provided in Chapter 4.

5.6 Second case study: Mine Pump Controller MPC

The Steam Boiler Controller SBC case study, presented in section 5.5, attempted to gradually exemplify the role of modes in simplifying scenario-based specifications. In this section, we further clarify this role by fleshing out the raw requirements of the Mine Pump Controller (MPC); a hybrid system similar to the SBC, but without any initial definitions of specific modes, as was the case with the SBC system studied in section 5.5.

The objective of the MPC case study in this section is to fully apply our procedure for eliciting mode-based specifications from raw requirements without any predefined modes. The reader should find the presentation of this case study as a natural progression from the previous case study in section 5.5, given that both case studies are drawn from same class of systems.

5.6.1 The Original Specifications of MPC

The water percolating into a mine is collected in a sump to be pumped out of the mine (See Figure 5.10). The water level sensors, D and E, detect when water is above a high and a low level respectively. A pump controller switches the pump on when the water reaches the high water level and off when it goes below the low water level. If the water cannot be pumped out, due to a failure in the pump, the mine must be evacuated within one hour.

The mine has other sensors (A, B, C) to monitor carbon monoxide, methane and airflow levels. An alarm must be raised and the operator informed within one second of any of these levels becoming critical, so that the mine can be evacuated within one hour. To avoid the risk of explosion, the pump must be operated only when the methane level is below a critical level.

Human operators can also control the operation of the pump, but within limits. An operator can switch the pump on or off if the water is between the low and high water levels. A special operator, the supervisor, can switch the pump on or off without this restriction. In all cases, the methane level must be below its critical level if the pump is to be operated.

Readings from all sensors, and a record of the operation of the pump, must be logged for later analysis.

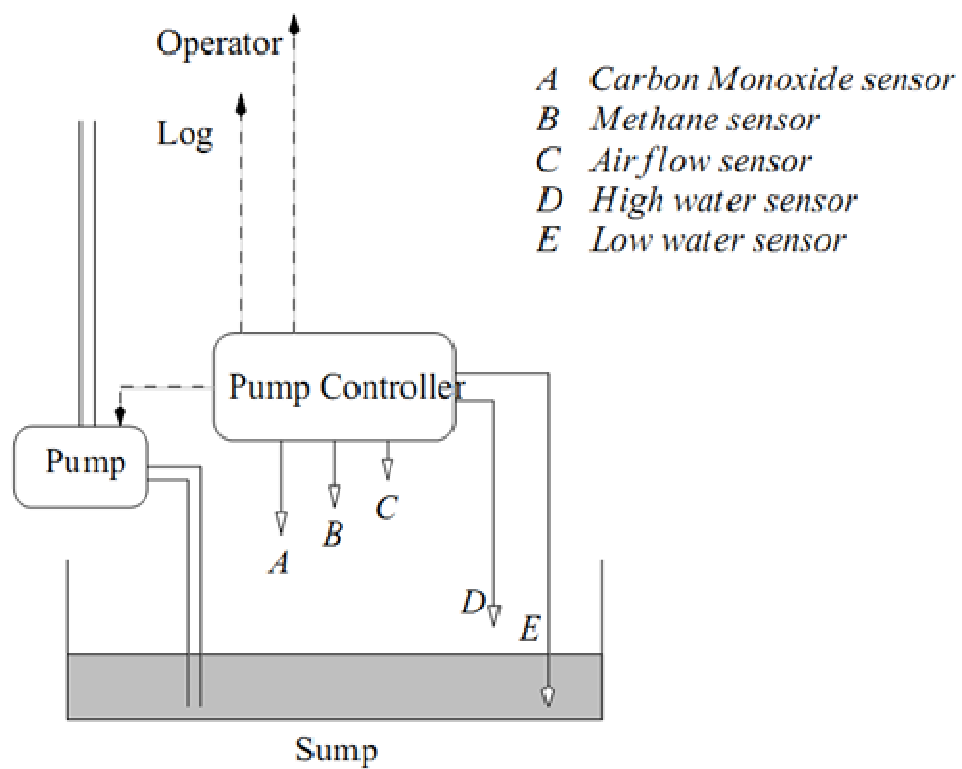


Figure 5.10: Schematic Diagram of the Mine Pump

From the informal description of the mine pump and its operations we obtain the following safety requirements:

1. The pump must not be operated if the methane level is critical.
2. The mine must be evacuated within one hour of the pump failing.
3. Alarms must be raised if the methane level, the carbon monoxide level or the airflow level is critical.

In the rest of this section, we take this textual specification and iteratively specify a structured model for Mine Pump Controller behavior, in terms of Modal Behaviors and Operational Behaviors. First we identify the domain variables of the Mine Pump Controller then we use these variables to describe the mode-classes and their corresponding mode-machines along with the scenarios descriptions in each mode.

5.6.2 Structuring the MPC Scenarios

In this section, we will apply our mode-based approach to specify the MPC's requirements, given in textual form in the previous section, in a structured form of modes and scenarios. The scenario in Figure 5.11 (adapted from [133]) is a direct extract from the MPC's textual requirements. We will use this scenario as an example of unstructured scenario specifications, and we will compare it to our structured scenario specifications we will develop in this section.

We will follow the procedure outlined in section 5.3.2. First, we identify the MPC's state variables that represent the basic "vocabulary" for further specifications. This involves identifying the input/output messages received/sent (respectively) by the MPC system (see Tables 5.6 and 5.7), and also identifying the state variables (see Table 5.8) and define them by directing predicating on those I/O messages. Then, all subsequent specifications will be defined in terms of those state variables and/or the I/O messages. More specifically, Modal Behaviors will be defined in terms of the state variables, and the (structured) scenarios' steps will be defined in terms of the the I/O messages.

5.6.2.1 Identifying the state variables

Figure 5.12 shows a context diagram of the MPC input/output relationship with other physical devices in the Mine Pump system. The events exchanged between the mine devices and MPC are codified as I/O messages. Tables 5.6 and 5.7 list the identified messages with a brief description of the event information they represent. We identified those messages from the given textual specifications (section 5.6.1).

Table 5.6: Messages received by the MPC

Message	Description
WATER_LOW	Sent to the MPC to when the current value of water level is LOW
WATER_MED	Sent to the MPC to when the current value of water level is MED
WATER_HIGH	Sent to the MPC to when the current value of water level is HIGH
METHANE_CRITICAL	Sent to the MPC when the methane level in air reaches a critical value
CARBON_CRITICAL	Sent to the MPC when the carbon-monoxide level in air reaches a critical value
AIRFLOW_CRITICAL	Sent to the MPC when the Airflow level reaches a critical value
PUMP_FAILED	Sent to the MPC when the Pump is failed
PUMP_READY	Sent to the MPC when the Pump is ready
MANUAL_PUMP_ON	Sent from the Operator desk to to the MPC to command it to switch the pump ON
MANUAL_PUMP_OFF	Sent from the Operator desk to to the MPC to command it to switch the pump OFF
SET_OPERATOR_NORMAL	Sent to the MPC when the current privileges of the human-operator changes to <i>normal</i>
SET_OPERATOR_SUPERVISOR	Sent to the MPC when the current privileges of the human-operator changes to <i>supervisor</i>

Table 5.7: Messages sent by the MPC

Messages	Description
SWITCH_PUMP_ON	sent to the pump controller to switch the pump ON
SWITCH_PUMP_OFF	sent to the pump controller to switch the pump OFF
TURN_ALARM_ON	sent to the Danger Light Pulp to turn it ON
TURN_ALARM_OFF	sent to the Danger Light Pulp to turn it OFF
OPERATOR_ACK	acknowledges a command from the operator (e.g., manually switching ON/OFF the pump)
OPERATOR_NACK	declines a command from the operator (e.g., manually switching ON/OFF the pump)
EVACUATE_MINE	signal to the operator desk to evacuate the mine

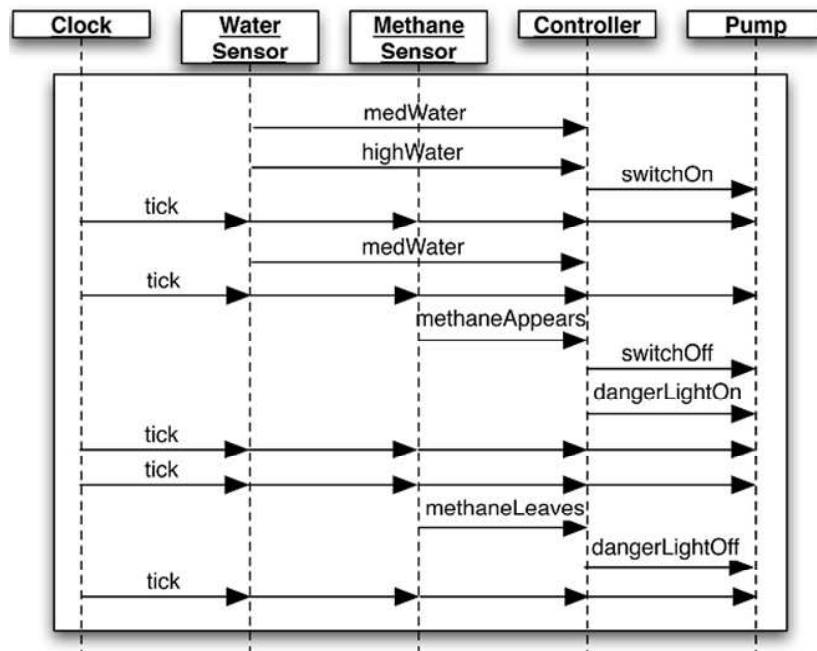


Figure 5.11: Naive scenario of MPC behavior (adapted from [133])

Given these I/O messages, we can then start to define the state information model of the MPC system. Recall that we have made a realistic analogy, in Chapter 4 (section 4.4), between the intuitive task of identifying state variables and the similar task of identifying *objects* in the Object Oriented Analysis and Design (OOAD) [80, 103] methodology⁵.

Table 5.8 details the state variables which we identified for the MPC system. We defined the variables by predicating directly on the I/O messages defined earlier.

Having defined the basic vocabulary (i.e., I/O messages and state variables) of the MPC system, we now start to elicit the different behaviors of the system – Modal Behaviors and Operational Behaviors.

5.6.2.2 Iteration #1

We start by applying the procedure described earlier in section 5.3.2. Recall that our procedure distinguishes between two types of behaviors: the Operational Behavior and Modal Behavior. Simply, a Operational Behavior is the behavior that takes place within a mode and does not execute across

⁵It is worth highlighting here that even though these two tasks (or methods) rely heavily on intuition and best practices and that they share the same purpose of *characterizing the system under development*, their outputs are actually quite different; the state variables characterize the system from black-box viewpoint, while objects or components imply clear-box viewpoint. Both are *static* characterizations, and still need further specifications for the *dynamic* aspect (modes and mode-machines for the former method, and component-level scenarios for the latter method.)

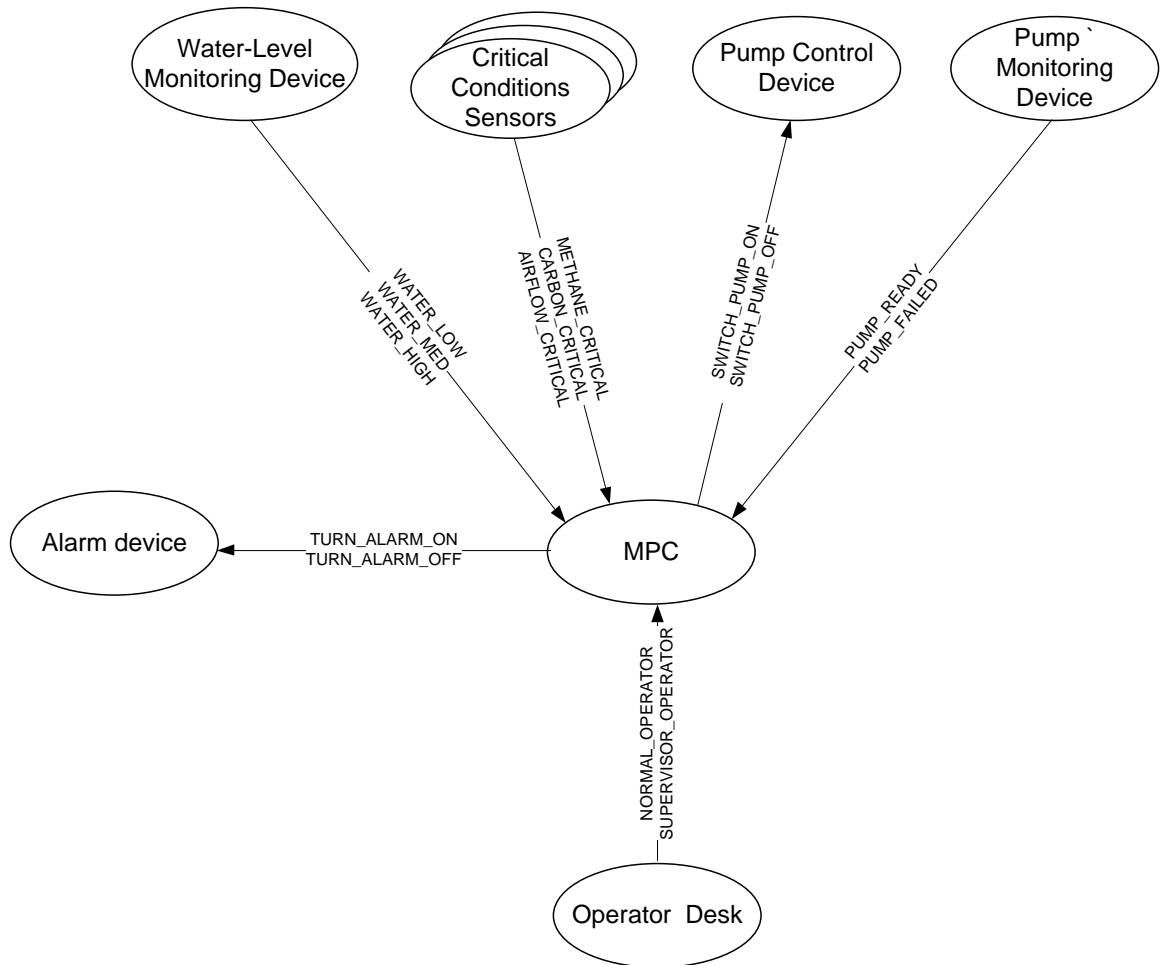


Figure 5.12: Context diagram of the Mine Pump system

Table 5.8: The identified state-variables of the Mine Pump Controller

predicate variable	Range values	Function specification
$s_PumpStatus$	<i>ON, OFF</i>	MANUAL_PUMP_ON $\Rightarrow (s_PumpStatus = ON)$ MANUAL_PUMP_OFF $\Rightarrow (s_PumpStatus = OFF)$
$s_PumpHealth$	<i>READY, FAILED</i>	PUMP_READY $\Rightarrow (s_PumpHealth = READY)$ PUMP_FAILED $\Rightarrow (s_PumpHealth = FAILED)$
$s_WaterLevel$	<i>LOW, MED, HIGH</i>	WATER_LOW $\Rightarrow (s_WaterLevel = LOW)$ WATER_MED $\Rightarrow (s_WaterLevel = MED)$ WATER_HIGH $\Rightarrow (s_WaterLevel = HIGH)$
$s_CriticalCond$	<i>NORMAL, CRITICAL</i>	CARBON_CRITICAL \vee AIRFLOW_CRITICAL $\Rightarrow (s_CriticalCond := CRITICAL)$
$s_MethaneCond^*$	<i>NORMAL, CRITICAL</i>	METHANE_CRITICAL \Rightarrow $s_MethaneCond := CRITICAL \wedge$ $s_CriticalCond := CRITICAL$
s_Alarm	<i>ON, OFF</i>	TURN_ALARM_ON $\Rightarrow (s_Alarm := ON)$ TURN_ALARM_OFF $\Rightarrow (s_Alarm := OFF)$

*Note that we kept the $s_CriticalCond$ and $s_MethaneCond$ separate because their specification is different: in case of $s_MethaneCond$ the mine must evacuate, in addition to firing Critical Condition alarm. So, as specified above in the table, $s_MethaneCond$ implies $s_CriticalCond$ but the reverse is not true.

modes belonging the same mode-class. On the other hand, the Modal Behavior determines the change (or the transition) between the different Operational Behaviors that the system executes, according to the current conditions of the system. In this section, we identify these behaviors for the MPC system, starting from its initial specifications described above. We will use the state variables in Table 5.8, wherever possible in the analysis.

The Modal- and Operational Behaviors of the MPC

After a first scan of the original specifications (Section 5.6.1), the MPC behavior is understood as controlling the water-level in the mine sump by pumping it out of the sump when it reaches a high limit (i.e., when $s_WaterLevel := HIGH$), and that this behavior can execute only if there are currently no critical conditions (i.e., when $s_CriticalCond := CRITICAL$). This can be analyzed as follows⁶:

- An Operational Behavior of the MPC is to *control the water-level*. – let us call it **Behavior-1**. This behavior does not affect other behaviors, so it is operational.
- An Operational Behavior of *controlling the danger alarm on appearance/absence of a critical condition* – let us call it **Behavior-2**. This behavior does not affect other behaviors, so it is

⁶Note that the decisions made on the behaviors types will be validated as we elicit more behaviors in the case study.

operational.

- And, a Modal Behavior of *controlling the pump activation whenever the methane level condition changes* – let us call it **Behavior-3**. This behavior affects **Behavior-1**, so it is a Modal Behavior.

A quick look at the original scenario in Figure 5.11 (reused from [133]), shows us that the Modal Behavior **Behavior-3** is mixed with the Operational Behaviors **Behavior-1** and **Behavior-2**. Our approach does not restrict combining multiple Operational Behaviors in the same scenario (though we recommend keeping them separate, if possible), but rather we clearly distinguish and separate modal and Operational Behaviors (which are mixed in Figure 5.11) for the reasons explained earlier in this chapter.

So far we have identified three behaviors; two of them are operational (**Behavior-1** and **Behavior-2**) and the third is modal (**Behavior-3**). The next step has to specify mode-classes for the Modal Behaviors and specify scenarios for the operational ones, as we see in the rest of this iteration.

Specifying the behaviors Behavior-1, Behavior-2 and Behavior-3: Following the procedure in section 5.3.2, we first specify the mode-classes for the Modal Behaviors and then try to run each Operational Behavior by those different modes (in each mode-class).

Consider MC_{meth} as the mode-class for the Modal Behavior **Behavior-3**. This behavior has two distinct contexts modeled as follows:

- $MC_{meth} = \{M_1^{meth}, M_2^{meth}\}$
- $M_1^{meth} \Rightarrow (s_MethaneCond = \text{NORMAL})$
- $M_2^{meth} \Rightarrow (s_MethaneCond = \text{CRITICAL})$

The next step is to *project* the Operational Behaviors identified so-far and specify their corresponding scenarios in each mode in MC_{meth} . For the Operational Behavior **Behavior-1**, we ran it by the two modes in MC_{meth} and this resulted in two scenarios for this behavior. We have pictorially “embedded” these scenarios in their *scoping* modes of MM_{meth} in Figure 5.13, and the detailed diagrams of these scenarios are also shown in Figure 5.14-(a) and Figure 5.14-(c).

Note that the scenario in Figure 5.14-(c) is actually a nil scenario. This is because the MPC system, itself, switches OFF while in M_2^{meth} and evacuation starts. So, effectively, the Operational Behavior **Behavior-1** has a single scenario in M_1^{meth} .

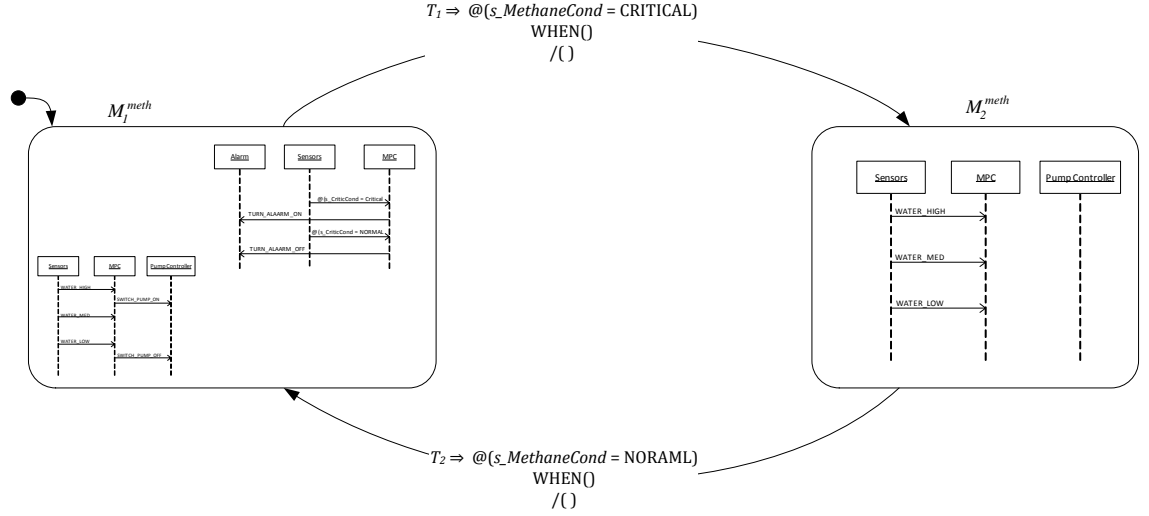


Figure 5.13: The MPC’s Modal Behavior *Behavior-3* (pump control on appearance and absence of Methane-level condition).

The Operational Behavior *Behavior-2* is modeled in Figure 5.14 (b) by a simple scenario to switch ON/OFF the alarm on the appearance of either the Carbon or Airflow critical conditions. Note that we modeled the events of these conditions by a single state variable $s_CriticCond$.

Before we finish this iteration, a genuine question here is that: why we did not use the $s_CriticCond$ to specify a Modal Behavior on its own? The answer is because the *mine evacuation* condition will not be part of either context of $s_CriticalCond$ is TRUE/FALSE, so this Modal Behavior will not be a mode-class.

5.6.2.3 Iteration #2

Now consider the next part of the original specifications in section 5.6.1: “if the pump failed and the water level is high, then the mine must be evacuated”. Let us call this behavior *Behavior-4*. This behavior represents change in the health status of the pump (ready/fail). According to the procedure in section 5.3.2, this behavior is of a modal type because it affects the Operational Behavior *Behavior-1*, “controlling the water-level”, and it does not affect any of the elicited Modal Behaviors. Let us model this behavior by the mode-class MC_{ps} , with two modes M_1^{ps} and M_2^{ps} (*ph* stands for *pump status*) such that:

- $MC_{ps} = \{M_1^{ps}, M_2^{ps}\}$
- $M_1^{ps} \Rightarrow (s_PumpHealth := READY)$

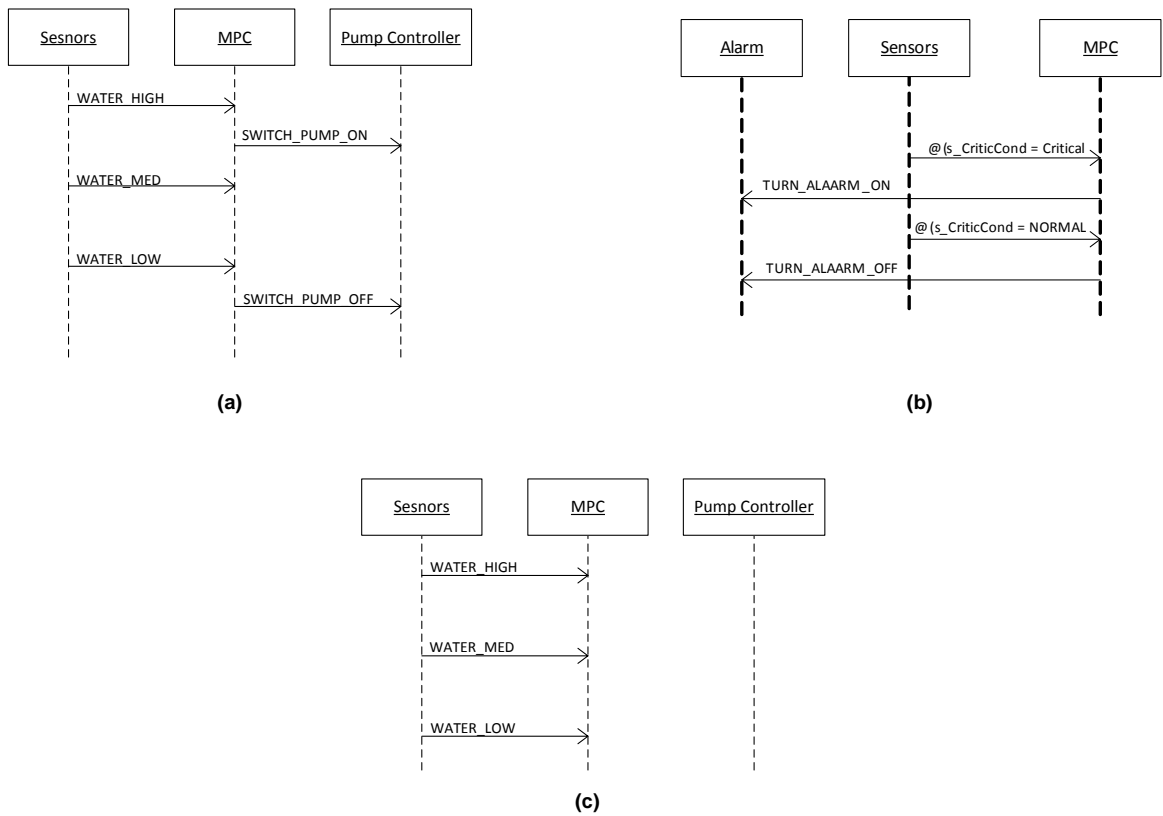


Figure 5.14: scenarios scoped by modes in the mode class MC_{meth} : (a) normal water-level scenario under scope of M_1^{meth} , (b) Alarm activation scenario under scope of M_1^{meth} too, (c) water-level control under scope of M_2^{meth}

(c) Water-level control scenario
Under a critical Methane level

(b) Alarm activation

(a) Water-level control scenario
under normal Methane level.

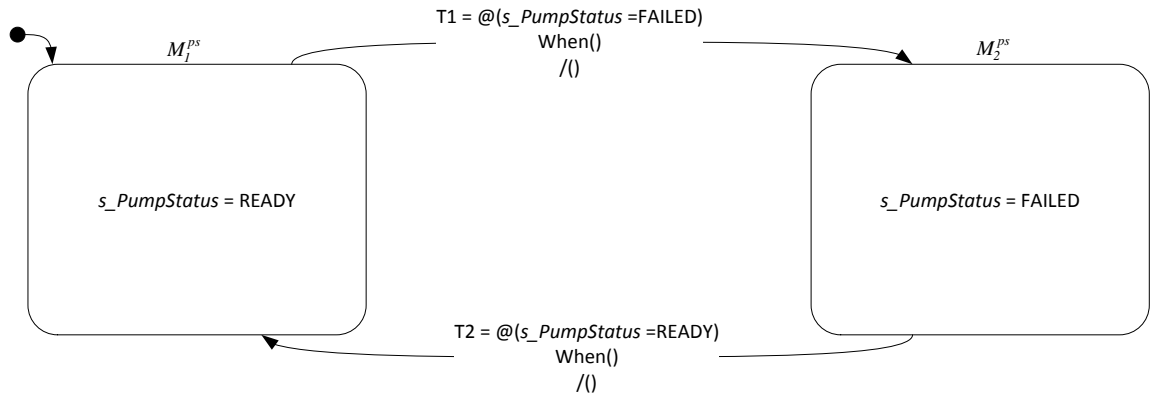


Figure 5.15: The MPC’s Modal Behavior *Behavior-4* of pump failure condition.

- $M_2^{ps} \Rightarrow (s_PumpHealth := FAILED)$

The mode-machine MM_{ps} in Figure 5.15 describes this behavior.

The MPC’s Modal Behavior of *Pump Failure Condition*.

Then we elicit the scenarios in each of the modes M_1^{ps} and M_2^{ps} in the mode-class MC_{ps} . Figure 5.16 below shows each of those scenarios.

In case of M_1^{ps} (pump is ready) the water-level control scenario is the same as the normal case (refer to *Behavior-1* that we studied earlier). However, in case of M_2^{ps} (pump was failed) the MPC sends a signal to evacuate the mine once the water-level reaches the high limit.

Initial analysis of *Behavior-4* reveals several ambiguities in the original specifications.

1. First, there is no device to which the MPC sends the “mine evacuation” alarm. We marked this unspecified object as unknown in the scenario in mode M_2^{ps} (Figure 5.16 (b)).
2. Second, there is nothing to say (in the original specifications) if the pump could be repaired or not (this indicated by the transition from M_2^{ps} back to M_1^{ps} in Figure 5.15), and nothing to say how the MPC is notified when the pump is repaired. For example, will the operator directly sends a “repair-done” message to the MPC; or shall the MPC query the pump monitoring devices in order to get this info?
3. The third ambiguity is that when the pump is repaired and the current mode is M_1^{ps} , shall the MPC switch the pump ON? Or shall the MPC keep the pump to OFF and expect the Operator to switch it ON as suitable?

We resolved these ambiguities by adding more behavior and messages to the specifications in Table 5.7 and to the scenarios in Figure 5.18.

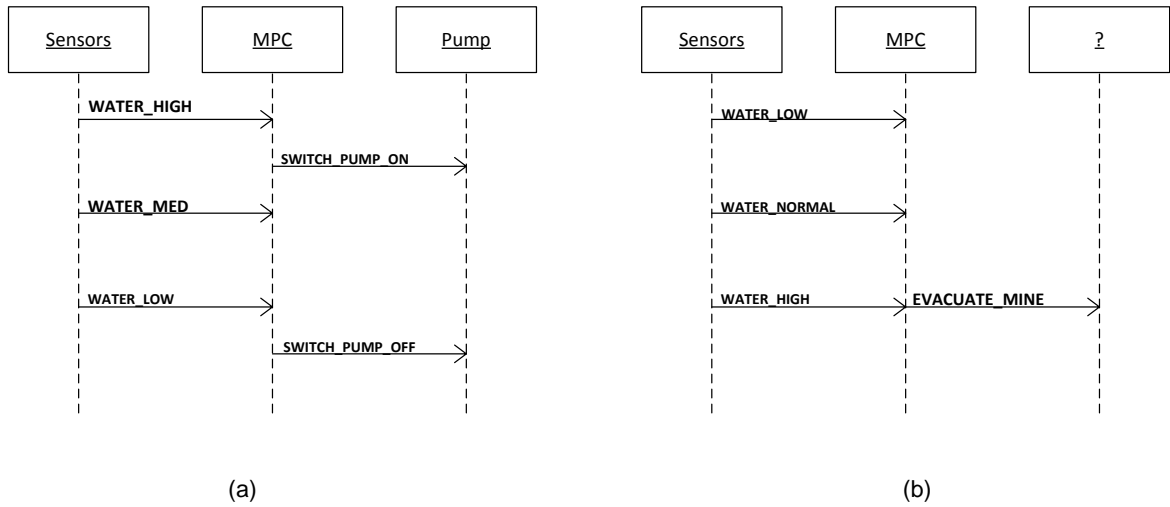


Figure 5.16: Scenarios scoped by MC_{ps} : (a) normal water-level control scenario in M_1^{ps} , (b) water-level control in M_2^{ps}

The final requirement in the original specifications – recalled here from Section 5.6.1 – states that “An operator can switch the pump ON or OFF if the water is between the low and high water levels. A special operator, the supervisor, can switch the pump on or off without this restriction. In all cases, the methane level must be below its critical level if the pump is to be operated.”

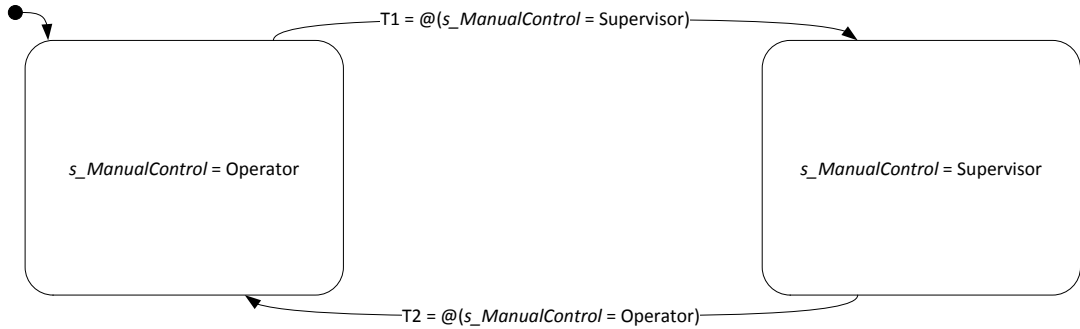
If we parse the textual description of this requirement, we can detect two distinct behaviors:

Behavior-5: the arbitrary switching of the pump ON or OFF by a human controller at the control panel, and

Behavior-6: the change of the authority-level of this human controller from a normal Operator to a Supervisor.

To reason about the types of behaviors in this requirement, we apply again our heuristic: Does any of those behaviors affect one or more existing Operational Behavior(s)? Could we elicit a variant of this behavior when it executes in different Modal Behaviors?

The initial observation is that **Behavior-6** affects **Behavior-5**. That is, the pump can be switched ON/OFF anytime in the case of Supervisor-level authority, however, this control is limited in the case of operator-level authority. Accordingly, we can conclude that **Behavior-6** is a Modal Behavior, while **Behavior-5** is Operational Behavior. We still need to see if this **Behavior-6** affects the other existing Operational and Modal Behaviors of the MPC. A quick check of the behaviors we elicited earlier in Section 5.6.2.2 will tell us that:



The MPC's Modal Behavior of Switching between an Operator and Supervisor level of manual control.

- The Modal Behavior **Behavior-6** does not affect **Behavior-1** ; whether an Operator or Supervisor is in charge, the MPC will control the water-level as specified in Figure 5.16 (a), even though this human controller switched the pump ON/OFF at specific times.
- The Modal Behavior **Behavior-6** does not affect the other Modal Behaviors **Behavior-3** and **Behavior-4**. This validates the mode-classes selection we have done, because Modal Behaviors are supposed to be independent and to affect the Operational Behaviors but not effect each other (otherwise we will have inter-dependencies that will distort the design regularity, that we seek).

Figure 5.17 below illustrates the mode-machine MM_{ca} ('ca' stands for control authority) of **Behavior-6**, where the manual control authority changes from Operator to Supervisor ('ca' stands for control authority).

The scenarios in Figure 5.18 below model the Operational Behavior **Behavior-6**. Figures 5.18 (a) and 5.18 (b) illustrate **Behavior-6** under scope of the modes M_1^{opr} and M_1^{opr} , respectively, where the operator's authority is Normal; Figure 5.18 (a) illustrates the two variants of the behavior in (1), and both of these scenarios are scoped by the mode in M_1^{opr} .

The scenarios of the Supervisor-level human controller are provided in Figures 5.18.(b).

5.6.2.4 Conclusion of the MPC case study

In this case study we took a further step further in our demonstration of the mode-based structuring of scenario specifications, and we partitioned the MPC behavior space into three mode-classes; *Methane critical condition* class, MC_{meth} , *pump health* class, MC_{ps} , and *operator authority-level* class, MC_{ca} .

To illustrate the resulting drawback of mixing modal- and Operational Behaviors let us consider

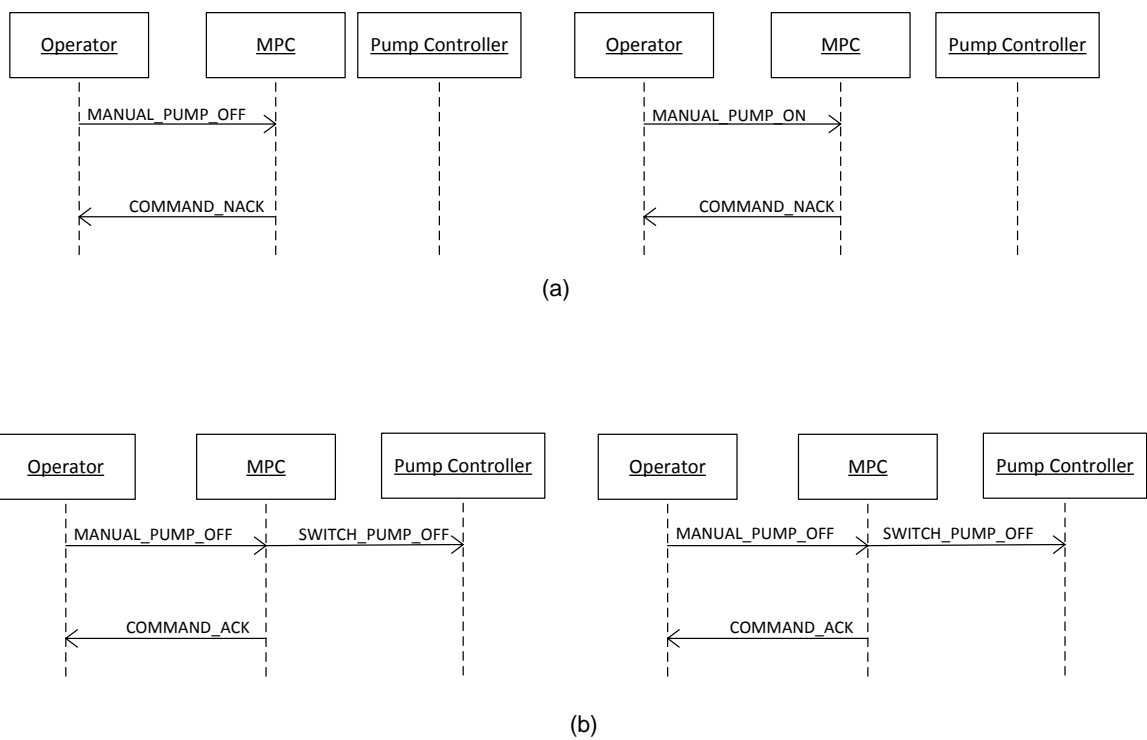


Figure 5.18: Water level control scenarios: (a) scenarios under scope of M_1^{ca} , (b) scenarios under scope of M_2^{ca}

the scenario in Figure 5.11. The scenario shows the Methane level as it crosses its critical level, while the *water-level decreases* from High to Medium. In order to add more flows, we then need either to extend this scenario or, to start another one such that it presents an additional cycle of *water-level increase* and introduces the *critical condition of methane* at the required point in this cycle. Nothing tells us which option is better. Moreover, we do not know which way to go first; to introduce the *critical condition* when the *water-level is High*, and then when it is *Low*, or the other way around? This shows us the maze we will end up in if we do not follow a structured way of specifying the scenarios here.

Another drawback of *ad hoc* specifications is that, assuming we could keep track of the possible scenarios, in order to specify the other cases where *the methane level becomes critical but the water-level is High*, and *when the water-level is Low*, we would need to add more scenarios to record these cases. Additionally, we would need to double all these scenarios to include the condition when a *pump failure* occurs, and we have to find out the various cases of different sequences of the occurrence of these conditions. Obviously, as the number of involved variables (and their ranges) increases, the number of scenarios explodes (possibly exponentially) in the valuations of these variables – pretty much similar to the state explosion problem, well known in formal verification techniques such as Model Checking.

By factoring the Modal Behavior out of Operational Behavior, we could elicit more behaviors that are not mentioned in the original specifications. For example, the Modal Behavior ***Behavior-4*** (elaborated in Figure 5.15) does not originally specify if the pump can be repaired at all, and what will be the correct action to take if the pump is repaired (e.g., will the MPC switch the pump ON if no critical condition is present) – leaving aside the handshaking mechanism, by which the operator communicates the repair information to the MPC.

5.7 Chapter Summary

We can say that this orthogonal relationship of Modal v.s. Operational Behavior allowed us to uncover behaviors and use-cases that would be hidden by the complexity of the required behavior. The Modal Behaviors provide a particular context within which the designers can safely exhaust all possible variants of the Operational Behavior, and this has led to uncovering those behaviors not spelt out in the original specifications. As we discussed earlier at several points, these contexts themselves are structured in a novel way (adopting a multi-view model) using modes and mode-classes.

This requirements specification approach would work best at early stage of system behavior

definition and specifications where designers and system analysts have general requirements that are possibly vague and might be also inconsistent.

This has been done by exposing technical questions to uncover requirements that would best be elicited at this early stage of development. We state that the description of each behavior can be (in simple cases) comprehended but ambiguities start to surface when those cases are merged together. For example, we can elicit a case of deadlock in Figure 5.18 (a), where the system has no way to go or to recover after it orders an evacuation.

Chapter 6

Synthesizing Automata Models From Structured Scenarios

In Chapter 3 we surveyed the state-of-the-art approaches for generating automata-based models from input scenario specifications. The surveyed approaches vary widely with respect to the algorithmic techniques used; the foundations of these techniques; the assumptions made on the input specifications; the form of the synthesized output models; etc. We also evaluated these approaches from those aspects and identified the shortcomings and the relevant gaps. In this Chapter, we attempt to fill in these gaps with a novel synthesis technique that accepts as input a structured scenario specifications, and iteratively produces, as output, a single state machine in the form of a Kripke Structure [82].

The structured scenario input is assumed to be specified according to the method we proposed in Chapter 5, and it includes a set of scenarios associated with a set of mode-classes. The mode-class specifications, part of the input specifications, are specified according to the approach proposed earlier in Chapter 4. Finally, the Kripke Structure is a popular form of transition system in paradigms, such as Model Checking [33], due to its simplicity, explicitly and amenability to analysis, together with its ability to generate code using standard off-the-shelf autocoding tools.

This chapter brings together all the work presented so-far in this thesis and uses it in a practical CASE¹ application. The model synthesis technique, which we present in this chapter, employs the approaches and methods presented in Chapters 4 and 5, as essential ingredients in the synthesis process.

¹Computer-Aided Software Engineering

6.1 New Process for Behavior Model Synthesis

Synthesis of automata models from partial specifications [83, 94, 129, 133, 145, 146] involves two major phases: translating the partial specifications to FSM models, and then merging these isolated models. The translation phase is typically straightforward and uses proprietary techniques for loop detection [83, 146]. The merge phase is more challenging due to difficulties in finding a common refinement model [133] that exhibits all the behaviors of the isolated models.

Our approach attempts to protect the specifications from partiality at the outset. It augments the scenario specifications with automata-based specifications in the form of mode machines. Disjoint subsets of scenarios are specified under the scope of different mode-machines. To synthesize a system model, we convert each subset of scenarios to FSMs and then integrate them by merging their scoping mode machines. This results in a common refinement model exhibiting all the behaviors of the scenarios.

In this section we present our proposed synthesis process to constructively and incrementally generate an automata-based behavior model, given a set of structured scenario specifications. Building on our proposed technique from chapter 5, to structure scenario-based requirements specifications, we use this (structured) form of scenarios as input to the synthesis process we present in this section. The output from this process is an integrated automaton that involves the execution of all behaviors implied by the input specifications, in addition to behaviors introduced during the synthesis process itself.

Our proposed process assumes designers' involvement when it encounters new decisions. Though the process automates the systematic and logical-reasoning related steps, it still requires the designer to resolve non-deterministic situations and to decide on new behaviors when they emerge during the synthesis process.

The process consists of a number of phases. Figure 6.1 is a detailed illustration of the process phases. We summarize here the basic ideas behind each phase, and provide a detailed descriptive discussion in subsequent sections.

The synthesis process goes through three phases. PHASE-1 is a preparation phase, and this involves two steps: first, the input specifications are checked for consistency. As shown in Figure 6.1, the specifications must be revisited and adjusted iteratively until the checks are passed. The second step translates scenarios to their FSMs counterparts using standard methods (c.f. [146]).

The actual synthesis steps start in PHASE-2. In this phase we, incrementally, merge the mode-machines in one single mode-machine. This is done by merging two mode-machines at a time such that they accumulate in one single mode-machine. New modes are likely to appear in the

resulting (system) mode-machine, due to (logical) intersections between modes from different mode-machines. During this merge process, the FSMs (translated from scenarios in PHASE-1) are processed so as to remove identical states that belong to different scenarios (that were scoped under different mode-machines). Another important processing of these FSMs is that the logical scoping relationship of 'mode X scopes state Y' is adjusted to accommodate the emerging (and the disappearing) modes due to the intersections.

At the end PHASE-2, we will have an intermediate automata model with a two-levels hierarchy. The higher level is a single mode-machine, which is the result of merging of all the mode-machines originally provided in the input specifications, along with additional modes that could possibly emerge from the merge operation. The lower level consists of those FSMs, that have been translated from the scenarios, originally provided in the input specifications. Each of those FSMs is scoped by one mode from the higher-level is a unique 1:1 relation.

In PHASE-3, the two hierarchies of automata models, generated in PHASE-2, will be integrated together in a single flat FSM that represents the system's behavior. More specifically, the individual FSMs will be connected by the transitions of the mode-machine (that constitute the higher-level hierarchy). During this "connecting FSMs" process, potential elaboration of the model is very likely to be happen, because it is unlikely that all (mode-level) transitions can connect FSMs. Automated checking tools (e.g. satisfiability checking tools SMT) will be helpful for checking if the mathematical predicate resulting from connecting two states (belonging to two different FSMs) to a mode-level transition does not violate any of the pre-specified constraints. The output of this phase is our final model in a form of flat FSM, that does not violate any of the pre-specified constraints. Also, this model is now ready for code generation, using off-the-shelf auto-coding tools.

In the rest of this section we go through each phase in detail and provide illustrative figures of the process when necessary. We also employ the ESFAS system [36] both as a running example to illustrate the presented concepts, and also as a case study at the end of this chapter for a complete demonstration of the synthesis process. An overview of the ESFAS system is provided in Chapter 2.

6.1.1 Process Phases

Given a structured scenario specification, we synthesize a standard FSM. We assume each scenario is specified in a standard Sequence Diagram (SD) form [103], with each of its messages annotated by *pre*- and *post*-conditions in the form of a vector of variables, similar to earlier approaches. Moreover, each message is labeled with an event predicate ℓ representing the conditions that triggered the message. Finally, each scenario is annotated with an identifier that refers to its

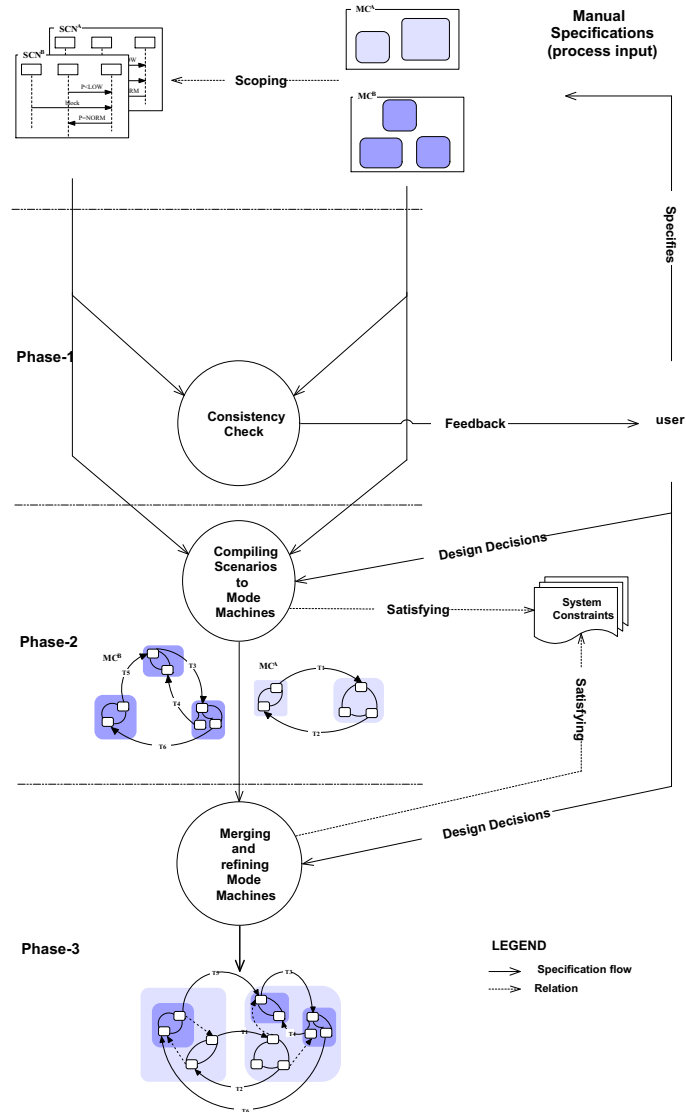


Figure 6.1: Behavior Synthesis Process (using micro-sized versions of ESFAS example.)

scoping mode. We also take into account the system constraints, denoted as C and formed as first-order formulas, specifying undesirable behaviors (e.g., safety properties). The synthesis process ensures that the generated models do not violate C .

Algorithm 1 describes the high-level procedure of the synthesis process. **Phase-1** checks consistency of input specifications and converts scenarios to their FSM counterparts. **Phase-2** merges the mode machines into a common refinement model MM_{sys} . The final phase, **Phase-3**, elaborates the mode-transitions in MM_{sys} to transitions connecting the scoped FSMs and results in the final state machine model FSM_{sys} . Algorithm 1 employs three auxiliary functions: **NEW()** for creating objects, and **ADD()** and **REMOVE()** for adding and removing one or more ‘elements’ to and from a ‘set’, respectively. We describe below each of these phases along with necessary explanations.

6.1.1.1 Phase-1: Preparing input specifications

There are two major steps in this phase: Step 1 is for consistency checking of the inputs and Step 2 is for conversion of the scenarios to FSMs. Step 1 checks the disjointness and space coverage (see Definition 2 in Chapter 4) properties of each mode machine. Moreover, consistency between the scenarios and their scoping modes is checked, such that the pre- and post-conditions in a scenario satisfy the predicate characterizing the scoping mode.

Step 2 employs standard techniques (cf. [146]) to convert the scenarios to their FSM counterparts. As an example, Figure 6.2(b) depicts the converted FSMs scoped by the MM^{pr} in the modes in the mode machine in the ESFAS system.

6.1.1.2 Phase-2: Space refinement (by merging mode-classes)

The basic hypothesis of our synthesis process is that *an integrated model can be obtained by a successive refinements of the state-space*. Every mode machine represents a view of the state-space partitioned into a set of contexts. The scenarios specified within these contexts are *partially refining* the state-space, because a scenario(s) is still likely to be partial even within the sub-space of its scoping mode. So, with several mode-machines – each with its own scoped scenario – we will have *several partial refinements* of the system state-space.

What we need to do next is to merge these partial refinements to find a common refinement model exhibiting all their behaviors. This merging process is formulated as follows:

Theorem 2. (*Mode-Classes Merging*). *Consider a space T partitioned by two mode-classes MC^a and MC^b . The intersection of MC^a and MC^b results in a new mode-class that is a common refinement such that $MC^a \prec MC^R$ and $MC^b \prec MC^R$. The two mode-classes MC^a and MC^b are said to be*

Algorithm 1 Procedure for the synthesis process.

Inputs:
(a) A set $ModeMachines = \{MM_1, MM_2, \dots, MM_m\}$ where $MM_i = \langle V, Q_{MC}^i, MC^i, M_0^i, \delta^i \rangle$; (b) A set of scenarios $SN = \{SN_1, SN_2, \dots, SN_n\}$; (c) A set of system constraints C
Outputs:
$FSM_{sys} = \langle T, t_0, \delta^{FSM} \rangle$, the integrated system FSM
Phase-1: Preparing specifications
1. $\forall i \in [1, m]$: Check disjointness of MC_i ; Check completeness of MC_i ; Check scenario are within their scoping modes; 2. $\forall i \in [1, n]$: Convert SN_i to FSM_i using the technique in [146]; ADD (FSM_i, FSM_{sys});
Phase-2: Space refinement (modes merge)
3. NEW ($MM_{sys} = \langle V, Q_{MC}^{sys}, MC^{sys}, M_0^{sys}, \delta^{FSM} \rangle$); while $ ModeMachines \geq 2$ do $\exists (MM_q, MM_r) \in ModeMachines$: $\forall (M_i^q, M_j^r) \in MC^q \times MC^r$: 4. ADD (M_i^q, M_j^r, MM_{sys}); 5. $\models_C (Q_i^q \wedge Q_j^r) \Rightarrow$ ADD (NEW (M_{ij}^{qr}), MC^{sys}) \wedge ADD ($(Q_{ij}^{qr} := (Q_i^q \wedge Q_j^r)), Q_{MC}^{sys}$); 6. $\exists t \in M_i^q : (t \models_C Q_{ij}^{qr}) \Rightarrow$ ADD (t, M_{ij}^{qr}) \wedge REMOVE (t, M_i^q); 7. /* repeat step 6 $\exists t \in M_j^r$ */ 8. $\exists t \in M_{ij}^{qr}$ combine identical states; $\forall k \in [1, MC^q]$ 9. $\exists M_k^q \rightarrow M_i^q \in \delta^q \Rightarrow$ $(\models_C (M_k^q \wedge \ell \wedge Q_{ij}^{qr})) \Rightarrow$ ADD (NEW ($M_k^q \rightarrow M_{ij}^{qr}$), δ^{sys}); 10. /* Repeat Step 9 $\exists M_i^q \rightarrow M_k^q$ */ 11. $\forall k \in [1, MC^r]$ /* Repeat Steps 9,10 for M_j^r */ 12. REMOVE ($MM_q, MM_r, ModeMachines$); ADD ($MM_{sys}, ModeMachines$); /* end while */
Phase-3: Modes Transitions Elaboration
$\forall (M_i^{sys}, M_j^{sys}) \in (MC^{sys} \times MC^{sys})$: $\forall (M_i^{sys} \xrightarrow{\ell} M_j^{sys}) \in \delta^{sys}$: 13. $\exists (t_s, t_d) \in (M_i^{sys} \times M_j^{sys}) : \models_C (t_s \wedge \ell \wedge t_d) \Rightarrow$ ADD (NEW ($t_s \xrightarrow{\ell} t_d$), δ^{sys}); 14. /* Repeat step 13 $\forall (M_j^{sys} \rightarrow M_i^{sys}) \in \delta^{sys}$ */

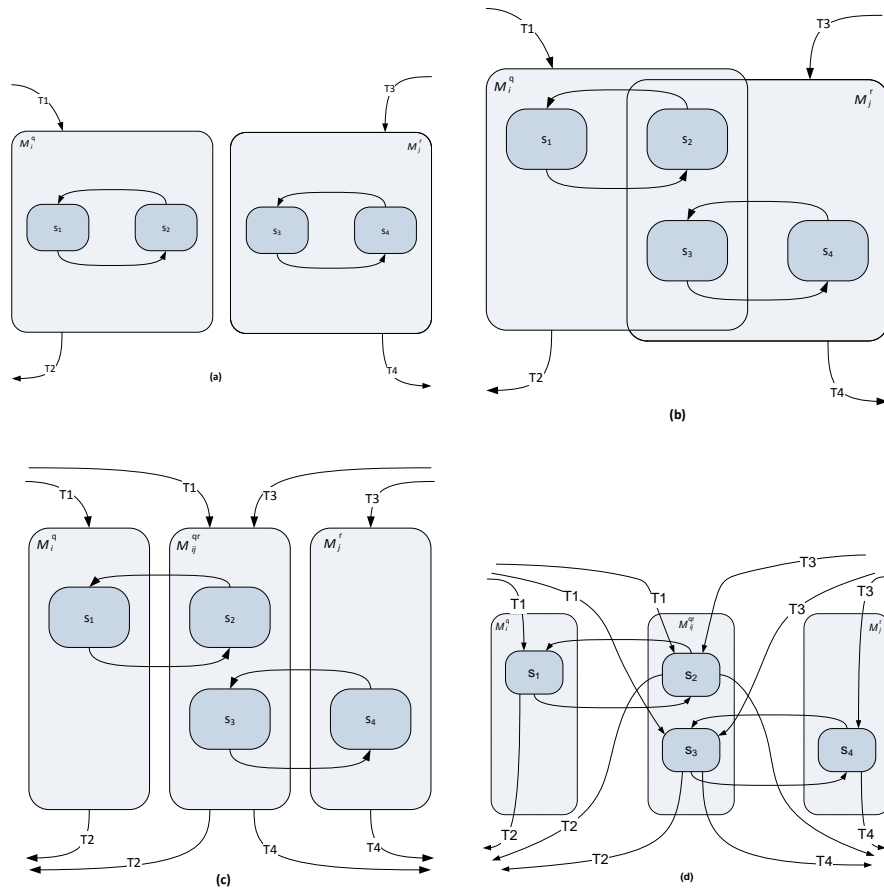


Figure 6.2: Illustration of the synthesis procedure (a) modes M_q^i and M_r^j overlapping (b) M_q^i and M_r^j after intersection (c) transition elaboration.

merged in one mode-class MC^R . This merge operation applies to any number of mode-classes partitioning T .

Proof. From the *coverage* and *disjointness* properties of a mode-class (Definition 2, Chapter 4), and since the mode-classes are partitioning the same space, then every mode in a mode-class fully overlaps with at least one mode (or possibly all modes) from any other mode-classes. Consequently, the intersection of a certain mode with its overlapping modes (from other mode-classes) results in a refinement of this mode. If we applied this intersection process to all modes in a certain mode-class, the result will be a refinement of this mode-class, which is also a common refinement of all other mode-classes, and this proves Theorem 2. \square

It is also deducible that for two mode-classes, where one of them is a refinement of the other, the result of merging them is identical to the refining mode. That is, if MC^b in Theorem 2 is a refinement of MC^a , then the merge result will be identical to MC^b .

Merging of mode-classes is the basic idea of space refinement in Phase-2 of the synthesis process. In Algorithm 1, Step 3 creates a mode machine object MM_{sys} , that will hold the common refinement of the input set $ModeMachines = \{MM_1, \dots, MM_m\}$. Steps 4-8 perform a merge of two modes M_i^q and M_j^r belonging to the mode-classes MC^q and MC^r , that, in turn, belong to mode machines MM_q and MM_r , respectively. Steps 9-11 elaborate the transitions connecting M_i^q with its peer modes in MC^q , and do the same with those transitions connecting M_j^r to its peer modes in MC^r . Figure 6.2 is a depiction of these steps. In Figure 6.2(a), the modes M_i^q and M_j^r are overlapping and a new mode M_{ij}^{qr} is created to scope the overlap area (Step 5). The states s_2 and s_3 are the only states in M_i^q and M_j^r satisfying both of these two modes (Step 6). In Figure 6.2(b), the newly created mode M_{ij}^{qr} has been assigned those states that are proven to satisfy it, and also it has inherited the transitions $T1$ to $T4$ (Steps 9-11) from its origin modes.

Steps 4-11 are iterated on all pairs of modes in the set $MC^q \times MC^r$, in an attempt to find a satisfiable overlap between every pair. This results in a merge of the mode-classes MC^q and MC^r , which is a *space refinement* of the mode machines MM_q and MM_r . Finally, Step 12 performs necessary data structure updates by replacing the merged mode machines, in the input set $ModeMachines$, with their refinement MM_{sys} , and loops back to merge MM_{sys} with another mode machine until the set Mode Machines has only one element, MM_{sys} . So, the result of the successive merging of all the input mode machines is accumulated in MM_{sys} .

6.1.1.3 Phase-3: Mode-transitions elaborations

As Figure 6.2(b) indicates, the output from Phase-1 is a model with only two levels of hierarchy: the mode machine MM_{sys} , at the higher-level, and the FSMs at the lower-level. Despite having the relationship *scoping*, they are still isolated in the sense of being a single executable automaton. The aim of Phase-3 is to elaborate the transitions between the modes to transitions between states in the FSMs. The rationale behind this step is, that when we move from abstract to concrete levels, a transition between two modes is actually summarizing several transitions between pairs of states that are local to those modes. These state transitions will have the same event condition, but possibly different guard conditions (which must be specified by designers to resolve non-determinism). Phase-3 attempts to find those pairs of states in the relevant modes (in the final common refinement model output from Phase-2) and assigns to them copies of the mode-transition(s), according to Definition 4 in Chapter 4.

The transitions elaboration should introduce *admissible transitions* only. Some of the transitions in the system are not admissible, either because of physical constraints in the runtime environment, or because of some design constraints imposed by the system design. We assume such constraints are included in the specifications C input to the process. Thus, in Steps 13-14, we create a state transition, only if the conjunction of the source state, the transition predicate and the destination state, is satisfiable with respect to the constraints C .

Finally, Figure 6.2(d) shows the result of elaborating the transitions from mode-level down to the state-level (assuming all the elaborated transitions are satisfiable according to the tests in Steps 13 and 14, Algorithm 1). Note that modes are shown in dotted lines because they have become redundant information in the integrated FSM model.

6.1.2 Observations and Discussion

In this section we described a procedure for the synthesis of an integrated system model from scenarios that are augmented with mode machines. The procedure is driven by a synergy between these two types of specifications (Section 4.1), where scenarios refine the mode machines, and the mode machines complement the scenarios' partiality and fragmentation.

The FSMs, counterparts of scenarios, are indirectly connected to construct the integrated system model. For example, in Figure 6.2(c) the states s_1 , s_3 are connected indirectly via a state where T1 originates. A similar case exists with T4 indirectly connecting s_2 and s_4 . In this sense, we have systematically linked the fragmented FSMs in one integrated model – the ultimate target of synthesis from partial specifications.

The emerging modes. A newly created mode, such as M_{ij}^{qr} in Step 5, may not have states (from FSMs) satisfying it, despite its predicate Q_{ij}^{qr} being satisfiable in the domain of discourse. Such modes will remain part of MM_{sys} and should be examined by the designer to see what behaviors they exhibit, in order to drive further elicitation of requirements.

Phases' execution order. Performing phases 2 and 3 in a different order should not lead to different results. If we independently elaborated the transitions in every mode machine to their scoped FSMs (i. e. performing Phase-3 on every mode machine) and then merged the mode machines successively (and combined the identical states as in Step 8) we will get the same result as if we followed Algorithm 1. As we will see in the Section 6.2.1, this *elaborate-then-merge* alternative is more intuitive for manual application of the synthesis procedure.

Algorithm complexity. Let EVL be the average time needed to evaluate a *wff*. Let SMC be the average time needed to find an interpretation satisfying a *wff*, with respect to the constraints C (in the sense of Satisfiability Modulo Theory (SMT) solvers). Assume **ADD()**, **REMOVE()** and **NEW()** execute in constant time.

Phase-1 depends largely on the response time of Theorem Provers, such as PVS, which have been used successfully to prove the disjointness and coverage properties [121].

Phase-2 effort is $m \cdot (msg \cdot EVL \cdot A + SMC \cdot B)$ where m is the number of mode machines, n is the average number of modes per machine, msg is the average number of messages per scenario, and A and B are arbitrary constants. Since $EVL \ll SMC$ by definition, the complexity is $O(m^n)$. Assuming two modes in MC_{sys} , each has two transitions between them.

Phase-3 has an effort of $S * |MC_{sys}| * 2 * SMC$ effort, where S is the total number of states in all FSMs, and complexity . Clearly, the driving factors in the overall effort are n and SMC . The number of modes per mode-class, n , is almost always small compared to the system size. This is because the main point of partitioning the state-space is to simplify it to a small number of sub-spaces – each is more abstract than the individual states. On the other hand, the SMC value depends on the response time of SMT solvers, which is expected to be reasonably small given the efficiency of existing SMT tools such as CVC3 and Yices.

6.2 Case Studies

6.2.1 ESFAS

We continue the design of the ESFAS system [36] in this section and use it as a case study to validate our mode-based design and synthesis approach. We also provide a summary of applying our

approach to the Steam Boiler case study [3]. The objective of these case studies is to demonstrate the potential of our approach in early requirements elaboration.

The ESFAS case study. The first specification step determines the mode-classes and their transition relation, forming the system’s mode-machines. We specified the MC^{pr} and MC^{sb} mode-classes along with their transition relations, depicted in Figure 4.5 (a). Next, we specified the scenarios within the scope of each mode. As an example, in Figure 4.5 (b), the internal FSMs within the modes of MM^{pr} have been converted from scenarios that are scoped by these modes.

Although, in theory, a mode-class can be any collection of predicates (as in Definition 2, Chapter 4) we purposefully specified the mode-classes that over-arch the system behavior from distinct aspects (which almost coincide with the mode-classes expressed in [36]). To this end, we found it a good design practice to minimize the variables involved in specifying each mode-class. If there are many system variables it is better to expand the specifications in breadth, *more classes*, rather than in depth, which would lead to *more modes in the same class*. This retains the simplicity which is important to this manual specification step, and avoids over-specifying a mode-class with details that should go in the scenarios themselves.

When we manually attempted the synthesis procedure in Algorithm 1 on the mode machines MC^{pr} and MC^{sb} , we found it difficult to manually merge the modes graphically as in Figure 6.2 because, when modes intersect, the transitions from different modes complicate the transition diagrams. As an alternative, we followed the *elaborate-then-merge* technique (see Section 4.3). Figure 6.3 shows the transitions elaboration for the individual machines. We independently elaborated the mode-transitions in MC^{pr} and MC^{sb} , and manually checked the admissibility of the introduced transitions. The inadmissible transitions are shown in red. For example, the red-colored transition T1 in Figure 6.3 (b) is inadmissible because the safety can not be blocked ($sb=on$), while the safety signal is already active.

We also elaborated behaviors for the requirement “the ESFAS must automatically reset the safety blocking button”. An initial design decision to satisfy this requirement is defining a timer $tr1$ to timeout the safety blocking. This resulted in adding the blue-colored transitions T6 and T7 in Figure 6.3 (b), and enhancing the FSMs in the modes M_1^{pr} and M_2^{pr} with similar transitions. However, when we introduced the transitions T2 and T3 in MC^{pr} (red-colored transitions in the upper side of Figure 6.3 (a)) we discovered the likely situation that, after the Operator blocks the safety to facilitate start-up, the pressure may rise to *norm* and come back to *low*, in a time shorter than *timeout* (due to some physical error that requires safety activation). A good decision is to let the safety unblock when $tr1=timeout$, or the pressure rises to *norm*. To this end, the transition T2, T3 (Fig 4(a)) are

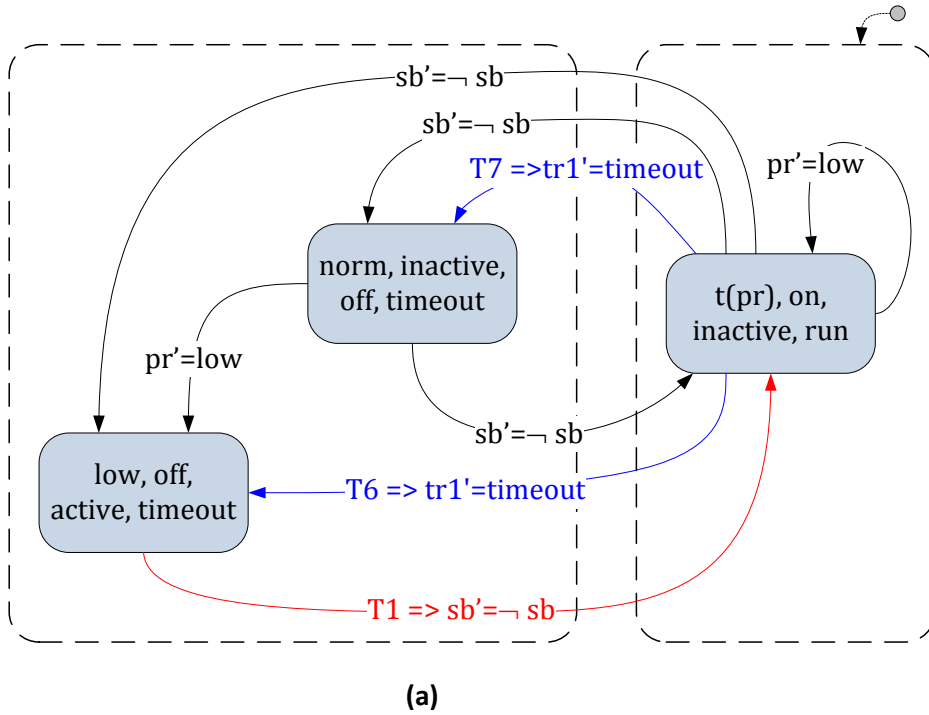
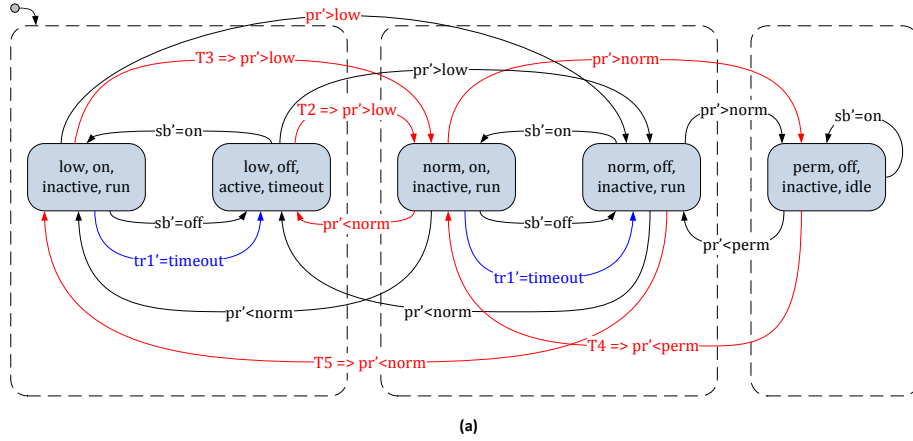


Figure 6.3: mode-machines in (a) MM^{pr} and (b) MM^{sb} are shown in dotted lines and their transition relations are elaborated to link the FSMs (solid lines). The red-color indicates the discovered prohibited transitions.

inadmissible.

Moreover, it was ambiguous as to how to decide the initial mode in MC^{sb} . If we decided it must be $sb=on$, where the Operator must block the safety before the ESFAS is powered-on, the Operator may get distracted and forget to do. In this case, ESFAS will immediately activate the safety on powering the plant on, and this might have undesirable consequences. On the other hand, if we decided $sb=off$ initially, then there is no mechanism to tell the ESFAS to wait for the Operator to press sb , who again might get distracted. A reasonable solution is to define another timer $tr2$ to allow the Operator enough time to block the safety to facilitate smooth start-up. Another yet issue illustrated by Figure 6.3 (b) is the deadlock state in M_1^{sb} ; this requires more elaboration of the scoped scenario to decide “what the system should do if the safety is activated,” particularly in case where Operator has missed pressing sb during cool-down to prevent unneeded safety activation. A final elaboration we could do is related to the red-colored T4 and T5 in Figure 6.3 (a). These transitions mean that the ESFAS can automatically block the safety, which is dangerous for the plant.

Note that the elaboration we achieved here has not been compared to existing approaches. We believe this would need a separate empirical study.

6.3 Related Work

A wide range of techniques have been proposed to automate the construction of behavioral models from partial specifications. To the best of our knowledge, no prior approach addresses the structuring of contexts in partial specifications in terms of coverage of behavior space and preventing inter-contexts traversing. However, the extraction of contextual information from a given unstructured specifications is often done [94, 129]. Below we summarize related work, classified by different aspects of this research area.

On modes. In the computing literature, there are two fundamentally different uses of the term “mode”: Modal Logic [30] and Hybrid Systems [93, 139]. The former is used to express the so-called necessity and possibility used in multi-valued logic. This use of the term “modes” is not related to the work presented here. The latter usage of ‘modes’ is to characterize different behaviors of a hybrid system. For example, a hybrid system exhibits different behaviors and each is characterized by differential equations. These behaviors are called system modes. The notion of mode that we use in this thesis is based on the ideas in [7] and has its origins in the theory of hybrid systems. Maraninchi and Remond [95] extended the synchronous language LUSTER with a mode construct which is, essentially, a discrete version of the Hybrid Automata. Another example in the Software Architecture

community, includes the use of modes in AADL [43], an Architecture Description Language, where a component behavior is mapped to a set of modes. Hirsch et al. [65] used modes to identify different structural topologies of components in a software architecture model. A few existing approaches attempted to formalize mode-based specifications techniques. Modechart by Jahanian and Mok [75] is a specification language based on RTL logic, however, they do not differentiate between a mode and a state, and they do not use the notion of mode-classes. Paynter [112] identified four possible ways to adopt modes in describing system behavior. Paynter adopted the non-exclusive option for describing modes (i. e. modes are not disjoint) to avoid proving of invalid properties (§2 in [112]). However, we believe that the option of disjoint modes combined with several mode-classes avoids these issues and also promotes the fundamental concepts of separation of concerns. Moreover, having several mode-classes for the same system allows a state to belong to several modes (but with each mode in a different mode-class) and achieves the same purpose as the non-exclusion option, adopted in [112].

On Synthesis from partial specifications. A common direction for addressing partiality is to enrich the machines, independently-generated from scenarios, with possible behaviors and delay refinement decisions until the merging phase. Uchitel et al. [136] (and improved upon in [134]) use special logic to capture the possible behaviors. Another range of techniques [6, 37, 94] infer behaviors from given specifications. A common denominator of these approaches is the use of bare scenarios without a structuring framework. This distracts the synthesis process from generating correct and structured models and focus on issues related to partiality itself. We consider partiality as a symptom of the main problem: the lack of a framework within which the partial specifications are described. This helped to relax the constraining assumption of that scenario must start at initial state, and to avoid the difficulties in finding a common refinement [133].

On combining different types of specifications. Sun and Dong [129] combine Live Sequence Charts, LSC, with Z specifications. They use predicate abstraction techniques to find predicates in LSCs. Although they extract abstract state-information, the extracted information does not guarantee scoping (coverage) of state-space as we do with mode-classes. Also, the mode-classes help designers to structure the specifications, and to provide feedback about the underspecified aspects of scenarios, and hence help in the earlier elicitation of requirements.

On the use of richer forms of scenarios. Other approaches accept more expressive forms of scenario than an SD as input. Uchitel et al. [137] synthesize behavioral models from Message Sequence Charts to detect implied scenarios. Whittle and Jayaraman [145] use the recent Interactions Overview Diagrams (IOD) in UML 2.0 [103] to generate Statechart-based designs. Though specified at a higher-level, these forms of scenario specification express control-flow information between basic

scenarios – an even more operational view of the system, with some control information, similar to that of flow-charts. The higher-level features in such specifications do not address coverage of the behavior space nor address structuring of contexts. Our approach is distinct by modularizing the specifications in the state-space, rather than relating them in a flowcharting structure. Although we assumed a simple form of scenarios in this thesis, more expressive forms can be also used as long as they are convertible to FSMs.

On automation. Many of the related approaches assume designers' involvement at some stage in the synthesis process. In Mäkinen and Systä[94], their MAS tool generates component-level behavior. Based on grammatical inference, MAS asks the designer trace questions in order to avoid undesirable generalizations. Trace questions may be quite hard for designers to understand if they are applied to system-level behavior. The same applies in Damas et al. [37], where the designer is presented with a whole scenario and to classify it as positive or negative. A related assumption about designer-involvement is also made in other inductive-learning approaches, such as [6]. When involving the designer, the key point is not to present too much information to be comprehended. In our approach, we involve the designer when a new mode, satisfying the constraints, is created. The new mode will be a conjunction of user-specified modes and will be intuitive enough to be comprehended. Also we involve the designer when a new (satisfying) transition is elaborated. The designer must examine these transitions that probably violate an important property yet to be specified (as we showed in the ESFAS case study). This adds to the designer's knowledge of the system behavior and allows incremental exploration of the state-space.

6.4 Conclusions

We have presented a design approach for behavior modeling and elaboration from partial requirements, accompanied with a novel synthesis technique to systematically merge partial specifications into an integrated system model. We formulated a mode-based behavior modeling framework as a direct interpretation of Parnas's ideas of *mode-classes*. We employed this framework to ameliorate the partiality issues in early requirements, at the outset. We augmented the scenarios with mode-based specifications so as to structure them and relate them in the system state-space. We detailed an algorithm that accepts as input *structured scenario specifications* and synthesizes an overall system model exhibiting the behaviors of the individual scenarios.

We suggest that our approach can be applied to other partial specifications, such as goals and properties. More generally, we envision mode-based design to be used as a design methodology similar

to Statecharts, without confusing the higher-level states with modes.

We plan to provide tooling support to allow scalability of our approach to real world applications. We plan to use existing reasoning tools, such as PVS, to check the consistency of specifications [121], and Satisfiability Module Theory solvers to automatically check when a new mode or transition is violating the system constraints. We also plan to have an interactive presentation for the designer showing the successive merge of machines and elaboration of transitions, allowing the designer to extend the system constraints based on the newly elaborated behaviors.

Chapter 7

Summary and Outlook

In the introduction to this thesis we have set out to research a methodical approach to *synthesize an analyzable system model from input requirements specifications*. We envisioned the outcome of this research helping to minimize the manual development effort of building such design models. Our approach contributes a new method of structuring requirements specifications that are given in the form of scenarios, plus a novel algorithm to synthesize automata-based models. This research work builds on and contributes to the existing research in the area of *design models synthesis*. In this concluding chapter we first recap the important assumptions, research targets and accomplishments that we have achieved during this research and reported in this thesis. Second we provide an outlook for the foreseen future of the research work in this thesis, in particular, and of design models synthesis research areas, in general.

7.1 Summary

In this section we summarize the contributions and accomplishment we achieved in this thesis, and where necessary we recap the assumptions and decisions we have made to develop our work, along with any known limitations.

7.1.1 *Synthesis-friendly specifications:*

To focus our research within this thesis, we decided to use scenario-based specifications (c.f., [26, 27, 28, 72, 103, 123]) as input to our synthesis method. The basic motivation was the intuitiveness of scenarios visual tools for engineers in communicating ideas and requirements, which

accounts for their wide use in industry. We envisioned this would facilitate the applicability of our approach in real industry cases. However, we believe our approach is applicable to other requirements specification forms too, and we emphasized this by decoupling both our proposed method (i.e., scenario structuring, in Chapter 4 and 5) and synthesis algorithm (Chapter 6) from the particular forms or constructs of scenarios – we will return to this point again shortly.

Due to the large number of research proposals in scenario-based specification, that covers various syntactical constructs and forms of scenarios, we had to select a subset of these constructs so that those constructs would be suitable for consumption by the synthesis process. To this end, we dedicated Chapter 2 to an analysis of a range of forms and constructs of scenario specifications. We surveyed representative approaches from each category of scenarios dialects, and we focused on those approaches with the widest use in research and industry. We used a specific criterion for this analysis and we considered every possible construct from the approaches surveyed. The criterion was based on *how far a scenario construct is amenable as input to synthesis*. Briefly, our conclusion was that:

1. Most scenario-based specification methods focus on programming-like constructs (e.g. decisions constructs, loop constructs, etc.),
2. Only basic constructs might be useful for processing by synthesis algorithms (e.g. Message Ordering and Environment Frame in MSC [69]).

Accordingly, we decided to use basic scenarios that are short and clear enough (i.e., no clumsy constructs) for human consumption, and we employed only the constructs (per the analysis in Chapter 2) that would facilitate the synthesis process.

7.1.2 Modes:

While simplifying the scenario form we used in our work (as discussed above), we were conscious too of the fact that such a simple form of scenarios, with only basic constructs, might not be suitable for larger scale systems. In such systems we would need too many scenarios to specify the requirements. For this reason, and others mentioned below, we identified the need for techniques or frameworks to deal with compound scenario specifications and this was one of the major challenges we faced in this thesis. To this end we proposed another approach based on *modes* (see Chapter 4 for detail) to “organize” multiple scenarios together in more compound forms. The motivation was manifold:

1. First, synthesizing an integrated system model from scenarios has encountered problems. During our survey and gap-analysis of existing synthesis approaches (Chapter 3) we noticed reports of difficulties in finding a *common refinement model* between multiple sub-models independently

generated from scenarios [133, 134]. We tied the problem to the specification stage where the requirements are given, typically, as a set of scattered scenarios.

2. Second, a scenario is inherently not concise. Even though a scenario is a simple artefact, capturing the requirements in scenario-based specifications becomes a tedious task for non-trivial systems, e.g. reactive systems. The scenario's sequence flow could get very long as there is nothing to guide the specifier when to stop a sequence (nor when to start it). Looking at the state-of-the-art (see Chapter 5) there is no existing conceptual model to guide the specification of scenarios.
3. Third, a scenario is a partial artefact. Scattered scenarios increase the likelihood of partiality in specifications. In our survey of scenario-composition approaches (see Chapter 5) we discussed the common theme of these approaches that employ a flowchart-like structure to link scenarios together. The well known approaches, such as High Level MSC (HMSC [72]) and Interactive Overview Diagrams (IOD [103, 145]) use similar programmatic constructs such as loops and decision points branching. Flowcharts have already been proven to be too simplistic when used at higher levels of specification [104, 105] and, as a consequence, they do not help to minimize the partiality problem.
4. Fourth, scenarios get complex very quickly in the presence of crosscutting concerns [34, 53, 79, 144]. The flow of design work, typically followed by designers, is to use the (initial) textual description of the system's behavior and start translate the text to an initial scenario. Typically, designers translate each part of the given textual requirements to a scenario, and they attempt to think of any other steps to add to the scenario until they run out of any more steps that would fit in that scenario. This will be repeated for all parts of the textual description. Next, designers start to think of "corner cases" and exceptions in the system's behavior, and continue to add more scenarios for each case until they comprehend all possible cases come to mind.
5. Finally, there is no guarantee capturing most behaviors of the system; hidden behavior cases could be easily overlooked. Leaving too many hidden behaviors maximizes partiality in the specifications, which is why scenarios are infamous-for causing this problem.

For all these genuine reasons, we worked for a conceptual framework to help capture requirements scenarios in a structured manner. We formulated this conceptual framework in Chapter 4, based on the classic notions of *modes* and *mode-classes* (c.f., [7, 36]). We used the basic foundations of theoretic

state-based automata and predicate logic concepts to define a formal model on this framework. We used a standard transition system to represent a mode-class, and we called this transition system a *Mode-Machine* (see Chapter 4 for detail).

Ultimately, the framework employs the (system's) state variables to structure the (system's) behavior space into "contexts". The notion of *context*, as we use it here, is simply a set of conditions, such that any behavior defined within this context must satisfy those conditions. Formally speaking, the designers can predicate directly on the state variables to define these contexts. The contexts are represented formally as a set of *modes* grouped in multiple (possibly overlapping) groups called *mode-classes*, such that each context (i.e., mode) is characterized by a predicate. We refer the reader to Chapter 4 for the details of the formulation of this framework.

7.1.3 *State variables:*

Our mode-based approach to structure scenarios assumes the provision of predefined state variables of the system under development. However, we were conscious of the fact that this assumption is not universally true. For example, in some systems, the domain experts can only speak about (expected) system behavior in terms of responses to events, rather than in terms of sequences of transitions between states. So we decided to improve our approach by keeping it rounded and self-contained.

To this end, we analyzed in Chapter 4 possible techniques to identify state variables from raw descriptions. We provided an analogy with the classic Object Oriented analysis method (OOAD [103]) and applied this practically in the case studies in Chapter 5.

7.1.4 *State-based versus Event-based automata:*

Deciding between state-based versus even-based automata models was also one of the choices we have made in this thesis. First, this was coincident with our dependency on state variables as an ingredient in our approach. Second, we have noticed most synthesis approaches we surveyed (in Chapter 3) were assuming event-based automata, and we decided to complement these approaches with our state-based one. Finally, our approach mainly targets reactive systems where environment variables are pervasive in the raw description of system behavior.

7.1.5 *Distinguished types of Behaviors:*

In Chapter 5 we have devised a methodical procedure to help designers use our approach to structure scenario-based specifications. In this procedure, we distinguished between two types of behaviors; the behavior specified as mode-machines and the behavior specified as scenarios. To allow this, we defined the notions *Modal Behavior* (to refer to behaviors defined as mode-machines) and *Operational Behavior* (to refer to behavior defined as scenarios). We applied this method on two well known case studies: the Steam Boiler Controller and the Mine Pump Controller. Our results show that the method allowed us to uncover implicit behaviors unheard-of in the original specifications, and to identify ambiguities in the given textual description.

7.1.6 *Dealing with Complexity:*

The idea of *mode* draws from classic concepts in hybrid systems theory, where a system behavior involves discrete and analog components. The complexity in hybrid systems comes originally from these irregularities in the overall (continuous) behavior. Modern hybrid systems are even more complex, such that system behavior might be completely different from one mode to another. Because traditional mathematical modeling (i.e., differential calculus) of continuous systems is not suitable for dealing with these radical change in behavior, introducing a discrete component (i.e., the transition system between hybrid behaviors) into the behavior model of these systems has greatly simplified the analysis of behavior – under the assumption that the system switches between these behaviors instantaneously (which is the same assumption underlying our digital systems).

Having this in mind, the idea of *mode-classes* is intended to simplify the complexity of crosscutting concerns in software systems, very much analogous to what discrete behavior has done to the complexities of hybrid systems. While the discrete behavior in hybrid systems has classified the irregular analog behavior into a set of regular analog behaviors, the *mode-classes* classify these modes further in a multi-dimensional way, such that each mode-class represents a *concern* in the system behavior.

7.1.7 *Synthesizing analyzable models:*

The second part of our development approach is the automation of the process of generating an analyzable design model from the captured specifications. This is achieved by synthesizing a standard state-based FSM that integrates all system behavior in a single model. We have formulated a refinement model of Mode Machines to allow the process of merging scenarios and assimilating them

into the mode-machines. We have sketched an iterative process of requirements elicitation and design model generation, to illustrate how these two development tasks integrate together. We also proposed a novel synthesis algorithm to implement this process. The algorithm operates as follows: first, it translates scenarios into state machines, then it integrates these FSMs with their corresponding mode machines to result in a flat FSM from each mode-machine, and finally it merges those FSMs into a single FSM that represents the overall behavior of the system. The resulting FSM is amenable to standard verification techniques, such as Model Checking, and can also be fed in to off-the-shelf tools to generate an implementation in a programming language, like C++ or Java. We have applied this algorithm to the ESFAS case study to illustrate the concepts involved, such as *merging of modes* and *transitions elaboration*.

7.1.8 Research validation and results

Early in this research project, we identified a set of case studies that allow us to demonstrate our research work. We used the ESFAS case study [36] and the Mine Pump Controller case study (See Appendix A) and both of these case studies have been used in model synthesis research work (c.f., [6, 11, 140, 86, 90, 135, 136, 133]). We also studied the Steam Boiler Controller system [2, 48] that has been used extensively in formal specifications of reactive systems.

Our findings, obtained from analysis in Chapters 4, 5 and 6, gave us solid indicators of the feasibility of the research we have done and have positively answered our target research questions which we have set out in Chapter 1. As expected, we came across some limitations, summarized below.

First, we aimed initially for a wider class of systems where this research can be applied, but this was not possible. During our analysis of the case studies, we found that this research would justify its application to reactive control systems (which already covers a wide range of real world systems), where environment variables are pervasive in the raw specifications. For example, data-intensive applications do not involve modal behaviors, so there could be better specifications approaches to model buffer management behavior and data processing. However, there are other non-control systems that have much modal behaviors, (but not necessarily state-based) to which we have not yet explored the applicability of our approach. For example, complex GUI and HCI systems, such as aircraft cockpit interface. Another candidate class of systems that would benefit from further study is *cyber-physical* systems that have a huge potential in future computing systems. Another known limitation in our approach is the lack of tool support to allow direct use and analysis of its complexity. Due to the large scope of tooling support for our approach, we kept tools development to the next phase of advancing this research work.

In the next section, we will address the hypothesis and research questions we have set in the introduction to this thesis, and detail our findings and answers we reached in this thesis.

7.2 Research Findings

This section is a retrospective on the research questions and the hypotheses we set out in the research plan of this thesis. We recall in our hypotheses and discuss how we have attempted to answer their associated question. We conclude on the results by drawing from our findings in the previous chapters. Chapter 1 details these hypotheses, the questions associated to each hypothesis and the research plan.

7.2.1 Hypothesis 1: The provision of distinct contexts of the system allows us to scope and structure partial specifications, such as scenarios, and enables better coverage of the requirements

This hypothesis has conjectured that one way to minimize the partiality problem in scenario-based specifications is to define, as a pre-specification step, a set of contexts that act as overarching conditions under which scenarios are then specified. In the following points we conclude on our findings that support this hypothesis.

7.2.1.1 How can the specification-framework reduce partiality of requirements

We attempted to address this question in Chapters 4 and 5. Our analysis of the partiality problem is that it is a symptom of a more basic problem: *lacking a design framework to guide and structure the specification task itself*. Another symptom of this basic problem is the so-called *implied behavior*, and this problem already has been addressed in other work (c.f., [135]). This analysis has influenced our research solution such that we researched a structuring framework for scenario specification. More specifically, we proposed a design method that augments scenarios with an automata-based model based on the idea of modes [7, 36] which provided the structuring ingredient in our approach.

The conceptual framework we described in Chapter 4 provided a formal foundation and a methodical technique to specify automata-based models (i.e., mode-machines) that augment scenarios. A mode-class partitions the state-space into contexts (or modes). This, in turn, divides the problem of comprehending the whole design space of behaviors into a clear confined scopes, and within each of context we specify shorter and maintainable scenarios. So, it will be easier for designers to exhaust all

possible scenarios with each context, separately, and we foresaw this to enrich the density of specifications.

We evaluated this approach using two case studies in Chapter 5. For instance, in the Steam Boiler Controller, we showed a 'naive' attempt to draw scenarios directly from the given textual specifications, and compared them to their shorter and more maintainable version specified with the contexts of the Failure Modes mode-class.

We captured a number of inconsistencies and gaps in the given specifications, as detailed in Chapter 5, that shows the potential of our approach to uncover behaviors that may not provided in the initial specifications that are typically collected from system experts. This supports our hypothesis of the foreseen potential of contexts to improve coverage of requirements and hence reducing partiality.

7.2.1.2 How can this framework relate the fragmented requirements together so as to facilitate their integration during the synthesis process

In addition to partiality, another issue with scenario specifications is fragmented nature of scenarios. This issue has been addressed in a number of existing research work (discussed in Chapter 5). We argued that the existing body of research work is using a flowchart-like structure to relate fragmented scenarios together in one model. Also we argued that flowchart structures are not suitable at the early stages of development when requirements exploration is pivotal to write comprehensive specifications.

Our mode-based structuring approach relates scenarios together via automata transition relation. This is unlike the flowchart approach assumed in the existing work. More interestingly, we emphasize on the main features of scenarios such as being short, maintainable and execute within confined scopes, and these are the main reason why scenarios are intuitive and hence they got popular in industry. Even though these features will increase fragmentation in scenarios, our approach augments this fragmentation. More specifically, the transition relationship which naturally exists in mode-machines will connect the scenarios (from different modes) together. One more advantage in our approach is the separating between the inter-scenario logic (which we refer to as Modal Behavior in Chapter 5) and the scenario flow (Operational Behavior) itself.

The Mine Pump Controller (MPC) case study (see Chapter 5) has demonstrated this technique. Applying our approach to specify the MPC requirements has resulted in many short Operational Behaviors modeled as scenarios. That was not a problem as we could systematically relate these scenarios together by their associated mode-machines.

The initial results we obtained and explained in Chapter 5 supports the claims in Hypothesis 1,

and we could see potential of the provision contexts to minimize partiality and fragmentation. Although we have not studied in this thesis the potential of this method to minimize other problems such as the emergence of implied behavior [135], we expect that contexts will offer a similar improvement as with partiality and fragmentation – given that all these problems in specifications are, in our opinion, symptoms of lacking a structuring framework.

7.2.2 Hypothesis 2: Structuring the partial requirements specifications would result in a more adequate input to synthesis techniques so as to avoid known issues in existing synthesis techniques.

In this hypothesis we have foreseen that the more structured the specifications are the more improved the synthesis process will be. We have specified two aspects of improvements: first, minimize the probability of lacking a cohesive model supposed to be built from the fragmented scenario specifications, and second, avoid the constraining assumptions about the input and/or the output specifications in the synthesis process. We will address each of these assumptions in the following discussion.

7.2.2.1 To what extent can the synthesis process guarantee the existence of a cohesive and integrated output model

The problem of generating non-cohesive output model (c.f., [133]) is that the input scenario include contradicting specifications. For example, if two separate scenarios include contradicting constructs (such as a scenario step or a condition) will prohibit the generation of a single model of the system.

There are a number of approaches (c.f., [85]) that could be used to code scenarios in formal models, such that they can be checked mechanically by software tools (such as Theorem Provers) to discover the contradictions and inconsistencies in the specifications. This process, however, is likely to be intractable for real world applications with several scenarios, in addition to the tedious task of coding scenarios themselves in those formal models. So, obviously, the best way to tackle this problem is to help preventing the contradicting specifications, or discover the contradiction before the synthesis process. We found that this preventive step is ideally done *during* the writing of the scenario specifications. Of course, the remit is on how this can be done.

In our mode-based approach to structure scenarios, the separation of concerns (or mode-classes) and the partitioning of each concern into contexts (i.e., modes in the same mode-class) are two

successive steps towards preventing the introduction of such contradictions in the scenarios. In our experience from the case studies in Chapter 5, we noticed that expressing actions or conditions without a context could be possible resulting in contradiction in the specification. Intuitively, a typical system may have two opposite actions that run under different conditions. The possible failure to synthesize a *Common Refinement Model* (i.e., an overall integrated model) in [133] has been corrected by going back to the input specifications and add a condition or adjust the scenario specifications. In our approach, we prevent this to happen by separating the Modal Behavior from the Operational Behavior (Chapter 4).

To conclude, the preventive approach we have taken in writing structured specifications has payed-off later during the synthesis process and has avoid contradictions that could arise during the specification task itself. In the example systems we studies in Chapter 5, we have found some ambiguities in the specifications of the SBC system and discovered it during the writing of the scenarios. In the example systems ESFAS we studied in Chapter 6, we have not encountered a situation where we could not integrate two mode-machines due to a contradiction, and if there were an issue it would have been discovered in the specification stage as has been done in Chapter 5.

7.2.2.2 What assumptions are necessary for the synthesis process, and how

There are three categories of assumptions in the synthesis process. First, assumptions on the type and form of input specifications. A prominent example of these assumptions is that a scenario starts from an initial state of the system. The second category is related to the output model, such as a specific dialect of formalism. The third category of assumptions is related to the processing steps synthesis process, such as the manual intervention to resolve emerging ambiguities during the synthesis.

Our findings are that these assumptions, such as constraining a scenario to start from initial state of the system, are not necessary for the synthesis process and that they are serving to a specific synthesis approach rather than being an essential requirement or the general synthesis. There are however unavoidable assumptions such as those related to the manual intervention, but they are not constraining the specifications form of technique.

Our experience with the case studies in Chapter 5, supports this finding as we have not put any assumptions or constraints on the form of scenario as long as it is convertible to a standard state machine. Also, we used a standard state machine to represent the output models, without any assumptions about dialects of the output model (e.g., Statechart as in [145] or MTS in [83, 133])

7.3 Outlook

The work in this thesis actually establishes a set of directions for future advancement. In the following discussion we highlight these directions and provide outlook vision of the prospect work in each.

Mode Machines semantics We assumed in this thesis that the mode-machines are flat and there is no concept of 'supermode' as with the case of superstate in hierarchical-state models such as Statemate. This is because of the many confusions arose from the concepts associated with these hierarchical-state models (e.g. a system can exist in two states at the same time, and an event can cross superstate boundaries, etc.)

The idea of hierarchical machines is itself a good idea to build concise models. So a candidate future work is to formalize semantics for mode machines models that would avoid the confusions found in existing hierarchical models.

Applying the approach to other partial specifications Although scenario-based specification type is most the popular, there are several other types of specifications that are emerging in the software engineering literature, such as Goals and *properties*. We plan to apply our approach these types too, and conclude with the common characteristics that would allows us to defined a unified method applicable (or customizable) to the different types of specifications.

Tool support One interesting direction to advance the research work in this thesis is to provide tooling of the methods described here. As a future work we plan to develop a tool to help in describing and analyzing the modal behavior modes and incrementally refine this behavior by lower-level specifications such as scenarios. We also plan to integrate this tool in one of the existing Satisfiability Modulo Theory SMT solvers such as *Yiece* and *CVC3* to be allows for in-process analysis and verification of the captured models, as well as helping to implement the relevant steps steps in our synthesis algorithm.

Bibliography

- [1] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In *ICALP*, pages 1–17, 1989.
- [2] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996. based on a Dagstuhl Seminar in June 1995.
- [3] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. The steam boiler case study: Competition of formal program specification and development methods. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications*, volume 1165 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0027228.
- [4] Luca Aceto, Anna Ingólfssdóttir, K Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification; ISBN 978-0-521-87546-2, pp. 300*, volume 35. ACM, New York, NY, USA, July 2010.
- [5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [6] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastián Uchitel. Learning operational requirements from goal models. In *ICSE*, pages 265–275, 2009.
- [7] Thomas A. Alspaugh, Stuart R. Faulk, Kathryn H. Britton, R. A. Parker, and David L. Parnas. Software requirements for the A-7E aircraft. Technical report, Naval Research Lab, Washington DC, 1992.
- [8] Rajeev Alur. A theory of timed automata, 1999.

- [9] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *ICSE*, pages 304–313, 2000.
- [10] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of msc graphs. *Theor. Comput. Sci.*, 331(1):97–114, February 2005.
- [11] Daniel Amyot, Sepideh Ghanavati, Jennifer Horkoff, Gunter Mussbacher, Liam Peyton, and Eric Yu. Evaluating goal models within the goal-oriented requirement language. *Int. J. Intell. Syst.*, 25(8):841–877, 2010.
- [12] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
- [13] J. Araújo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdogan. Early aspects: The current landscape. (CMU/SEI-2005-TN-xxx), 2005. <p>n/a</p>.
- [14] Robert L. Baber, David L. Parnas, Sergiy A. Vilkomir, Paul Harrison, and Tony O’Connor. Disciplined methods of software specification: A case study. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC’05) - Volume II - Volume 02*, ITCC ’05, pages 428–437, Washington, DC, USA, 2005. IEEE Computer Society.
- [15] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. *SIGPLAN Not.*, 36(5):203–213, May 2001.
- [16] Nicolas Baudru and Rémi Morin. Synthesis of safe message-passing systems. In *Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science*, FSTTCS’07, pages 277–289, Berlin, Heidelberg, 2007. Springer-Verlag.
- [17] P. Beynon-Davies, C. Carne, H. Mackay, and D. Tudhope. Rapid application development (rad): an empirical review. *Eur. J. Inf. Syst.*, 8(3):211–223, September 1999.
- [18] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning communicating automata from mscs. *IEEE Trans. Softw. Eng.*, 36:390–408, May 2010.
- [19] Yves Bontemps. Realizability of scenario-based specifications. Diplôme d’études approfondies, Facultés Universitaires Notre-Dame de la Paix, Institut d’Informatique (University of Namur, Computer Science Dept),, rue Grandgagnage, 21, B5000 - Namur (Belgium), September 2003.
- [20] Yves Bontemps and Patrick Heymans. Turning high-level live sequence charts into automata. In *Proc of Scenarios and State Machines: Models Algorithms and tools, 24th International Conf. on Software Engineering, May 2002*, ACM, 2003.

- [21] Yves Bontemps, Patrick Heymans, and Pierre-Yves Schobbens. From live sequence charts to state machines and back: A guided tour. *IEEE Trans. Software Eng.*, 31(12):999–1014, 2005.
- [22] Yves Bontemps, Patrick Heymans, and Pierre-Yves Schobbens. Lightweight formal methods for scenario-based software engineering. In Stefan Leue and Tarja Systä, editors, *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 575–575. Springer Berlin / Heidelberg, 2005.
- [23] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundam. Inf.*, 62(2):139–169, February 2004.
- [24] Daniel Brand and Pitro Zafropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, April 1983.
- [25] Julius R. Buechi. On a Decision Method in Restricted Second-Order Arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.
- [26] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [27] Robert L. Campbell. Categorizing scenarios: a quixotic quest? *SIGCHI Bull.*, 24(4):16–17, 1992.
- [28] Robert L. Campbell. Will the real scenario please stand up? *SIGCHI Bull.*, 24(2):6–8, 1992.
- [29] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
- [30] B.F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.
- [31] Chia-Chu Chiang and Joseph E. Urban. Incremental elicitation and formalization of user requirements through rapid prototyping via software transformations. In *COMPSAC*, pages 240–245, 1996.
- [32] Mary Beth Chrissis, Mike Konrad, and Sandy Shrum. *CMMI Guidelines for Process Integration and Product Improvement*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [33] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

- [34] Siobhán Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. In *ICSE*, pages 5–14, 2001.
- [35] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, 1998.
- [36] P.-J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proceedings of the 15th international conference on Software Engineering*, pages 315–323, Baltimore, Maryland, United States, 1993. IEEE Computer Society Press.
- [37] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Trans. Software Eng.*, 31(12):1056–1073, 2005.
- [38] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Form. Methods Syst. Des.*, 19(1):45–80, July 2001.
- [39] Jules Desharnais, Marc Frappier, Ridha Khédri, and Ali Mili. Integration of sequential scenarios. *IEEE Trans. Software Eng.*, 24(9):695–708, 1998.
- [40] Pierre Dupont. Incremental regular inference. In *ICGI*, pages 222–237, 1996.
- [41] Rüdiger Ehlers. Generalized rabin(1) synthesis with applications to robust system synthesis. In *Proceedings of the Third international conference on NASA Formal methods, NFM'11*, pages 101–115, Berlin, Heidelberg, 2011. Springer-Verlag.
- [42] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL: formal object-oriented language for communicating systems*. Prentice Hall, 1997.
- [43] H. P. Feiler, B. Lewis, and S Vestal. The sae architecture analysis and design language (aadl) standard. In *IEEE RTAS Workshop*, 2003.
- [44] Joao M. Fernandes, Simon Tjell, Jens Baek Jorgensen, and Oscar Ribeiro. Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In *Proceedings of the Sixth International Workshop on Scenarios and State Machines, SCESM '07*, pages 2–, Washington, DC, USA, 2007. IEEE Computer Society.
- [45] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, L. Finkelstein, and Michael Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.

- [46] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering, FOSE '07*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.
- [47] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
- [48] Marie-Claude Gaudel, Pierre Dauchy, and Carole Houry. A formal specification of the steam-boiler control problem by algebraic specifications with implicit state. In *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*., pages 233–264. Springer-Verlag, 1996.
- [49] Blaise Genest. Compositional message sequence charts (cmscs) are better to implement than mscs. In *TACAS*, pages 429–444, 2005.
- [50] Blaise Genest, Dietrich Kuske, and Anca Muscholl. A kleene theorem and model checking algorithms for existentially bounded communicating automata. *Inf. Comput.*, 204(6):920–956, 2006.
- [51] Blaise Genest, Dietrich Kuske, and Anca Muscholl. On communicating automata with bounded channels. *Fundam. Inform.*, 80(1-3):147–167, 2007.
- [52] Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level mscs: Model-checking and realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006.
- [53] Geri Georg, Indrakshi Ray, and Robert B. France. Using aspects to design a secure system. In *ICECCS*, pages 117–, 2002.
- [54] A. Gill. *Applied algebra for the computer sciences*. Prentice-Hall series in automatic computation. Prentice-Hall, 1976.
- [55] Wolfgang Grieskamp and Nicolas Kicillof. 6th international workshop on scenarios and state machines: Models, algorithms, and tools (scesm07). In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 129–130. IEEE Computer Society, 2007.
- [56] Radu Grosu, Ingolf Krüger, and Thomas Stauner. Hybrid sequence charts. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC '00*, pages 104–, Washington, DC, USA, 2000. IEEE Computer Society.

- [57] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.
- [58] D. Harel and H. Kugler. Synthesizing state-based object systems from lsc specifications. Technical report, Jerusalem, Israel, Israel, 1999.
- [59] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
- [60] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In Hans-Jürgen Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Formal Methods in Software and Systems Modeling*, volume 3393 of *Lecture Notes in Computer Science*, pages 309–324. Springer Berlin / Heidelberg, 2005.
- [61] David Harel and Amnon Naamad. The state machine semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.
- [62] Matthew Hause. The sysml modelling language. In *Fifteenth European Systems Engineering Conference*, 2006.
- [63] Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Software Eng.*, 6(1):2–13, 1980.
- [64] Jesper G. Henriksen, Madhavan Mukund, K. Narayan Kumar, Milind A. Sohoni, and P. S. Thiagarajan. A theory of regular msc languages. *Inf. Comput.*, 202(1):1–38, 2005.
- [65] Dan Hirsch, Jeff Kramer, Jeff Magee, and Sebastián Uchitel. Modes for software architectures. In *EWSA*, pages 113–126, 2006.
- [66] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.
- [67] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1979.
- [68] Pei Hsia, Jayarajan Samuel, Jerry Gao, David Kung, Yasufumi Toyoshima, and Cris Chen. Formal approach to scenario analysis. *IEEE Softw.*, 11(2):33–41, March 1994.
- [69] ITU. Recommendation z.120 : Message sequence chart (msc), 1996.

- [70] ITU. Recommendation z.120 : Annex b, 1998.
- [71] ITU. Recommendation z.120 :(11/99) : Msc 2000., 1999.
- [72] ITU. ITU-T Recommendation Z.120 – Z series: Languages and general software aspects for telecommunication systems - formal description techniques (FDT) - message sequence chart (MSC), April 2004.
- [73] Michael Jackson. Specializing in software engineering. *IEEE Software*, 16(6):119–121, 1999.
- [74] Michael Jackson. The name and nature of software engineering. In *Lipari Summer School*, pages 1–38, 2007.
- [75] Farnam Jahanian and Aloysius K. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. Software Eng.*, 20(12):933–947, 1994.
- [76] Shmuel Katz, Mira Mezini, Christa Schwanninger, and Wouter Joosen, editors. *Transactions on Aspect-Oriented Software Development VIII*, volume 6580 of *Lecture Notes in Computer Science*. Springer, 2011.
- [77] Raman Kazhamiakin, Marco Pistore, and Marco Roveri. Formal verification of requirements using spin: A case study on web services. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 406–415, Washington, DC, USA, 2004. IEEE Computer Society.
- [78] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, July 1976.
- [79] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *AOSD*, pages 87–98, 2009.
- [80] Haim Kilov, William Harvey, and Reino Kurki-Suonio. Fundamentals of object-oriented specification and modeling of collective behaviors. In *Object-Oriented Behavioral Specifications*, volume 371 of *The Kluwer International Series in Engineering and Computer Science*, pages 101–120. Springer US, 1996. 10.1007/978-0-585-27524-6-7.
- [81] Nicos Komninos, Marc Pallot, and Hans Schaffers. Special issue on smart cities and the future internet in europe. *Journal of the Knowledge Economy*, pages 1–16, 2012. 10.1007/s13132-012-0083-x.

- [82] Saul Kripke. Semantic considerations on modal logic. *Acta philosophica fennica*, 16:83–94, 1963.
- [83] Ivo Krka, Yuriy Brun, George Edwards, and Nenad Medvidovic. Synthesizing partial component-level behavior models from system specifications. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*, pages 305–314, Amsterdam, The Netherlands, 2009. ACM.
- [84] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, November 1995.
- [85] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From mscs to statecharts. In *Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, pages 61–71, Norwell, MA, USA, 1999. Kluwer Academic Publishers.
- [86] Renaud De Landtsheer, Emmanuel Letier, and Axel van Lamsweerde. Deriving tabular event-based specifications from goal-oriented requirements models. In *RE 2003*, 2003.
- [87] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210, 1988.
- [88] Edward A. Lee. Finite state machines and modal models in ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009.
- [89] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Fluent temporal logic for discrete-time event-based models. *SIGSOFT Softw. Eng. Notes*, 30(5):70–79, September 2005.
- [90] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastián Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Autom. Softw. Eng.*, 15(2):175–206, 2008.
- [91] Stefan Leue, Lars Mehrmann, and Mohammad Rezai. Synthesizing room models from message sequence chart specifications. Technical Report 98-06, ECE Department, University of Waterloo, Ontario, April 1998.
- [92] Markus Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theor. Comput. Sci.*, 309(1-3):529–554, 2003.

- [93] J. Lunze and F. Lamnabhi-Lagarriue. *Handbook of Hybrid Systems Control: Theory, Tools, Applications*@bookvan2000introduction, title=*An Introduction to Hybrid Dynamical Systems*, author=*Van Der Schaft, A.J. and Schumacher, J.M.*, isbn=*9781852332334*, lccn=*99049518*, series=*Lecture Notes in Control and Information Sciences*, url=*http://books.google.ie/books?id=vdVSAAAAMAAJ*, year=*2000*, publisher=*Springer* . Cambridge University Press, 2009.
- [94] Erkki Mäkinen and Tarja Systä. MAS - an interactive synthesizer to support behavioral modeling in uml. In *ICSE*, pages 15–24, 2001.
- [95] Florence Maraninchi and Yann Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3):219–254, 2003.
- [96] James Martin. *Rapid application development*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1991.
- [97] S. Mauw and M. A. Reniers. Operational semantics for msc'96. *Comput. Netw.*, 31(17):1785–1799, June 1999.
- [98] John McManus and Trevor Wood-Harper. A study in project failure. *British Computer Society*, 2008.
- [99] Andreas Metzger and Stefan Queins. Early prototyping of reactive systems through the generation of sdl specifications from semi-formal development documents. In *SDL Forum Society; University of Wales*, 2002.
- [100] Andreas Metzger and Stefan Queins. Model-based generation of sdl specifications for the early prototyping of reactive systems. In *Proceedings of the 3rd international conference on Telecommunications and beyond: the broader applicability of SDL and MSC*, SAM'02, pages 158–169, Berlin, Heidelberg, 2003. Springer-Verlag.
- [101] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
- [102] Rémi Morin. Recognizable sets of message sequence charts. In *STACS*, pages 523–534, 2002.
- [103] OMG. UML 2.0 specification, Sept. 2003.
- [104] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

- [105] David Lorge Parnas. Designing software for ease of extension and contraction. In *ICSE*, pages 264–277, 1978.
- [106] David Lorge Parnas. Software aspects of strategic defense systems. *Commun. ACM*, 28(12):1326–1335, 1985.
- [107] David Lorge Parnas. Some theorems we should prove. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 155–162. Springer-Verlag, 1994.
- [108] David Lorge Parnas. The use of mathematics in software engineering. In *ICFEM*, page 1, 2000.
- [109] David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148, 2010.
- [110] David Lorge Parnas and Harald Würges. Response to undesired events in software systems. In *ICSE*, pages 437–446, 1976.
- [111] Oscar Pastor and Juan Carlos Molina. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
- [112] Stephen Paynter. Real-time mode-machines. In *Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 90–109. Springer-Verlag, 1996.
- [113] Dennis K. Peters and David L. Parnas. Requirements-based monitors for real-time systems. *IEEE Trans. on Software Engineering*, 28:146–158, 2000.
- [114] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.
- [115] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.
- [116] Bruno R. Preiss. *Data structures and algorithms with object-oriented design patterns in C++*. John Wiley & Sons, Inc., 1999.
- [117] Colm Quinn, Sergiy A. Vilkomir, David Lorge Parnas, and Srdjan Kostic. Specification of software component requirements using the trace function method. In *ICSEA*, page 50, 2006.

- [118] Awais Rashid. Aspect-oriented requirements engineering: An introduction. In *RE*, pages 306–309, 2008.
- [119] Door Michel Adriaan Reniers. Message sequence chart: Syntax and semantics. Technical report, Faculty of Mathematics and Computing, 1998.
- [120] James E. Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language reference manual*. Addison-Wesley-Longman, 1999.
- [121] John M. Rushby and Mandayam K. Srivas. Using pvs to prove some theorems of david parnas. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 163–173. Springer-Verlag, 1994.
- [122] M. Samek. *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*. Elsevier Science, 2002.
- [123] Bikram Sengupta and Rance Cleaveland. Executable requirements specifications using triggered message sequence charts. In *Proceedings of the Second international conference on Distributed Computing and Internet Technology, ICDCIT'05*, pages 482–493, Berlin, Heidelberg, 2005. Springer-Verlag.
- [124] Mary Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer (STTT)*, 4:1–7, 2002. 10.1007/s10009-002-0083-4.
- [125] Mary Shaw. Writing good software engineering research papers: minitutorial. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 726–736, Washington, DC, USA, 2003. IEEE Computer Society.
- [126] Hesham Shokry and Mike Hinchey. Model-based verification of embedded software. *IEEE Computer*, 42(4):53–59, 2009.
- [127] Stephane S. Some. Beyond scenarios: Generating state models from use cases, 2001.
- [128] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. *SIGSOFT Softw. Eng. Notes*, 30(4):1–6, May 2005.
- [129] Jun Sun and Jin Song Dong. Design synthesis from interaction and state-based specifications. *IEEE Trans. Software Eng.*, 32(6):349–364, 2006.

- [130] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering, ICSE '99*, pages 107–119, New York, NY, USA, 1999. ACM.
- [131] Shmuel Tyszberowicz and Amiram Yehudai. Observ a prototyping language and environment. *ACM Trans. Softw. Eng. Methodol.*, 1(3):269–309, July 1992.
- [132] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Behaviour model synthesis from properties and scenarios. In *Proceedings of the 29th international conference on Software Engineering, ICSE '07*, pages 34–43, Washington, DC, USA, 2007. IEEE Computer Society.
- [133] Sebastián Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Software Eng.*, 35(3):384–406, 2009.
- [134] Sebastián Uchitel and Marsha Chechik. Merging partial behavioural models. In *SIGSOFT FSE*, pages 43–52, 2004.
- [135] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Detecting implied scenarios in message sequence chart specifications. In *ESEC / SIGSOFT FSE*, pages 74–82, 2001.
- [136] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Behaviour model elaboration using partial labelled transition systems. In *ESEC / SIGSOFT FSE*, pages 19–27, 2003.
- [137] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85, 2004.
- [138] Joseph E. Urban. Software prototyping and requirements engineering. Technical report, Rome Laboratory, RL/C3CB 525, Brooks Road Griffiss AFB, NY 13441-4505, June 1992.
- [139] A.J. Van Der Schaft and J.M. Schumacher. *An Introduction to Hybrid Dynamical Systems*. Lecture Notes in Control and Information Sciences. Springer, 2000.
- [140] Axel van Lamsweerde and Emmanuel Letier. From object orientation to goal orientation: A paradigm shift for requirements engineering. In Martin Wirsing, Alexander Knapp, and Simonetta Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *Lecture Notes in Computer Science*, pages 153–166. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24626-8-23.

BIBLIOGRAPHY

- [141] A.J. Van Schouwen. *The A-7 Requiriments Model: Re-examination for Real-time Systems and an Application to Monitoring Systems*. CRL Report n. 242. McMaster University, 1992.
- [142] Simona Vasilache and Jiro Tanaka. Bridging the gap between analysis and design using dependency diagrams. In *SERA*, pages 407–414, 2005.
- [143] Michael von der Beeck. A comparison of statecharts variants. In *FTRTFT 1994 (LNCS 863)*, pages 128–148, 1994.
- [144] Jon Whittle and João Araújo. Scenario modelling with aspects. *IEE Proceedings - Software*, 151(4):157–172, 2004.
- [145] Jon Whittle and Praveen K. Jayaraman. Synthesizing hierarchical state machines from expressive scenario descriptions. *ACM Trans. Softw. Eng. Methodol.*, 19(3), 2010.
- [146] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *ICSE*, pages 314–323, 2000.
- [147] Kang Kyo C Wood, David P. A classification and bibliography of software prototyping. Technical Report CMU/SEI-92-TR-013, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.
- [148] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [149] T. Ziadi, L. Helouet, and J.-M. Jezequel. Revisiting statechart synthesis with an algebraic approach. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 242 – 251, may 2004.